

Getting Started with Altera's DE2 Board

This document describes the scope of Altera's DE2 Development and Education Board and the supporting materials provided by the Altera Corporation. It also explains the installation process needed to use a DE2 board connected to a computer that has the Quartus[®] II CAD system installed on it.

Contents:

Purpose of the DE2 Board

Scope of the DE2 board and Supporting Material

Installation and USB-Blaster Driver

Using the DE2 Board

Altera's DE2 Development and Education Board has been developed to provide an ideal vehicle for learning about digital logic and computer organization in a laboratory setting. It uses the state-of-the-art technology in both hardware and CAD tools to expose students to a wide range of topics covered in typical courses. The power of the board is such that it is also highly suitable for a variety of design projects as well as for the development of sophisticated digital systems. In addition to the DE2 board and the associated software, Altera provides supporting materials that include tutorials, laboratory exercises, and interesting demonstrations.

1 Purpose of the DE2 Board

University and college courses on the design of logic circuits and computer organization usually include a laboratory component. In a modern curriculum, the laboratory equipment should ideally exemplify state-of-the-art technology and design tools, but be suitable for exercises that range from the simple tasks that illustrate the most basic concepts to challenging designs that require knowledge of advanced topics. From the logistic point of view, it is ideal if the same equipment can be used in all cases. The DE2 board has been designed to provide the desired platform.

2 Scope of the DE2 Board and Supporting Material

The DE2 board features a powerful Cyclone® II FPGA chip. All important components on the board are connected to the pins of this chip, allowing the user to configure the connection between the various components as desired. For simple experiments, the DE2 board includes a sufficient number of switches (of both toggle and pushbutton variety), LEDs, and 7-segment displays. For more advanced experiments, there are SRAM, SDRAM, and Flash memory chips, as well as a 16 x 2 character display. For experiments that require a processor and simple I/O interfaces, it is easy to instantiate Altera's Nios II processor and use interface standards such as RS-232 and PS/2. For experiments that involve sound or video signals, there are standard connectors provided on the board. For large design projects, it is possible to use USB and Ethernet connections, as well as the SD memory card. Finally, it is possible to connect other user-designed boards to the DE2 board by means of two expansion headers.

Software provided with the DE2 board features the Quartus II web edition design tools. It also includes a simple monitor program that allows the student to control various parts of the board in an easily understandable manner. There are also several applications that demonstrate the utility of the DE2 board.

Traditionally, manufacturers of educational FPGA boards have provided a variety of boards and the CAD tools needed to implement designs on these boards. However, there has been a paucity of supporting materials that could be used directly for teaching purposes. Altera's DE2 board is a significant departure from this trend. In addition to the DE2 board, Altera Corporation provides a full set of associated exercises that can be performed in a laboratory setting for typical courses on logic design and computer organization. In effect, the DE2 board and the available exercises can be used as a ready-to-teach platform for such laboratories. Of course, the DE2 board is also likely to be suitable for exercises that have been developed for other hardware platforms and can be ported to the DE2 platform.

3 Installation and USB-Blaster Driver

The DE2 board is shipped in a package that includes all parts necessary for its operation. The only essential parts are the 9-volt power adapter and the USB cable. There is also a protective plexiglass cover that may be used in the laboratory environment to protect the board from accidental physical damage.

Plug in the 9-volt adapter to provide power to the board. Use the USB cable to connect the leftmost USB connector (the one closest to the power switch) on the DE2 board to a USB port on a computer that runs the Quartus II software. Turn on the power switch on the DE2 board.

The computer will recognize the new hardware connected to its USB port, but it will be unable to proceed if it does not have the required driver already installed. The DE2 board is programmed by using Altera's USB-Blaster mechanism. If the USB-Blaster driver is not already installed, the New Hardware Wizard in Figure 1 will appear.



Figure 1. Found New Hardware Wizard.

Since the desired driver is not available on the Windows Update Web site, select No, not this time in response to the question asked and click Next. This leads to the window in Figure 2.

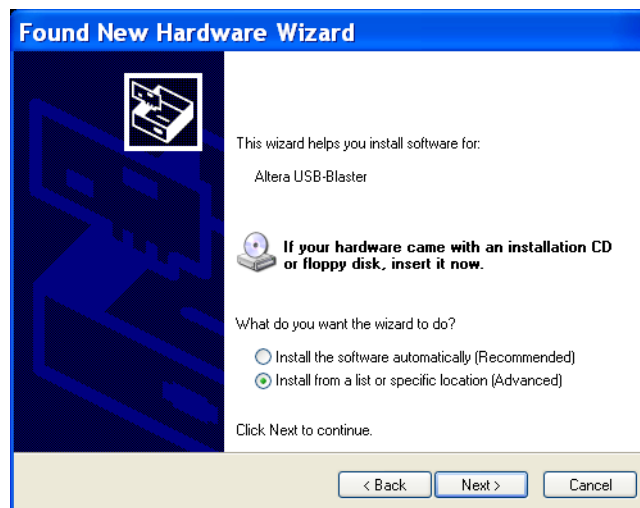


Figure 2. The driver is found in a specific location.

The driver is available within the Quartus II software. Hence, select Install from a specific location and click Next to get to Figure 3.



Figure 3. Specify the location of the driver.

Now, choose **Search for the best driver in these locations** and click **Browse** to get to the pop-up box in Figure 4. Find the desired driver, which is at location *altera\quartus50\drivers\usb-blaster*. Click **OK** and then upon returning to Figure 3 click **Next**. At this point the installation will commence, but a dialog box in Figure 5 will appear indicating that the driver has not passed the Windows Logo testing. Click **Continue Anyway**.

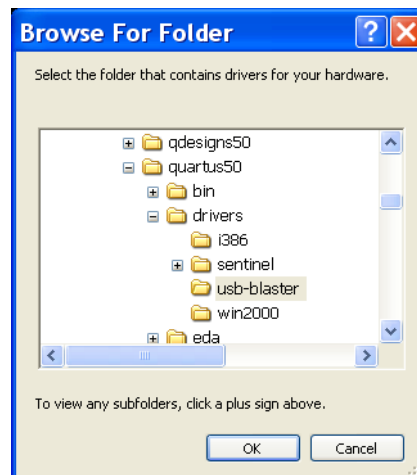


Figure 4. Browse to find the location.



Figure 5. There is no need to test the driver.

The driver will now be installed as indicated in Figure 6. Click Finish and you can start using the DE2 board.



Figure 6. The driver is installed.

4 Using the DE2 Board

The DE2 board is used in conjunction with the Quartus II software. A reader who is not familiar with this software should read an introductory tutorial. There are three versions of the tutorial:

- *Quartus II Introduction Using Verilog Design*
- *Quartus II Introduction Using VHDL Design*
- *Quartus II Introduction Using Schematic Design*

These tutorials cover the same aspects of the Quartus II software; they differ only in the design entry method that is used. They illustrate the entire process of implementing a design targetted for the DE2 board.

Detailed information about the DE2 board is given in the *DE2 User Manual*, which describes all of the features of the board. It also describes a Control Panel utility which allows the user to write/read data into/from various components on the board in a simple and direct manner. The user is encouraged to explore and make use of this utility.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Quartus II Introduction Using Verilog Design

This tutorial presents an introduction to the Quartus[®] II CAD system. It gives a general overview of a typical CAD flow for designing circuits that are implemented by using FPGA devices, and shows how this flow is realized in the Quartus II software. The design process is illustrated by giving step-by-step instructions for using the Quartus II software to implement a very simple circuit in an Altera FPGA device.

The Quartus II system includes full support for all of the popular methods of entering a description of the desired circuit into a CAD system. This tutorial makes use of the Verilog design entry method, in which the user specifies the desired circuit in the Verilog hardware description language. Two other versions of this tutorial are also available; one uses the VHDL hardware description language and the other is based on defining the desired circuit in the form of a schematic diagram.

The last step in the design process involves configuring the designed circuit in an actual FPGA device. To show how this is done, it is assumed that the user has access to the Altera DE2 Development and Education board connected to a computer that has Quartus II software installed. A reader who does not have access to the DE2 board will still find the tutorial useful to learn how the FPGA programming and configuration task is performed.

The screen captures in the tutorial were obtained using the Quartus II version 5.0; if other versions of the software are used, some of the images may be slightly different.

Contents:

- Typical CAD flow
- Getting started
- Starting a New Project
- Verilog Design Entry
- Compiling the Design
- Pin Assignment
- Simulating the Designed Circuit
- Programming and Configuring the FPGA Device
- Testing the Designed Circuit

Computer Aided Design (CAD) software makes it easy to implement a desired logic circuit by using a programmable logic device, such as a field-programmable gate array (FPGA) chip. A typical FPGA CAD flow is illustrated in Figure 1.

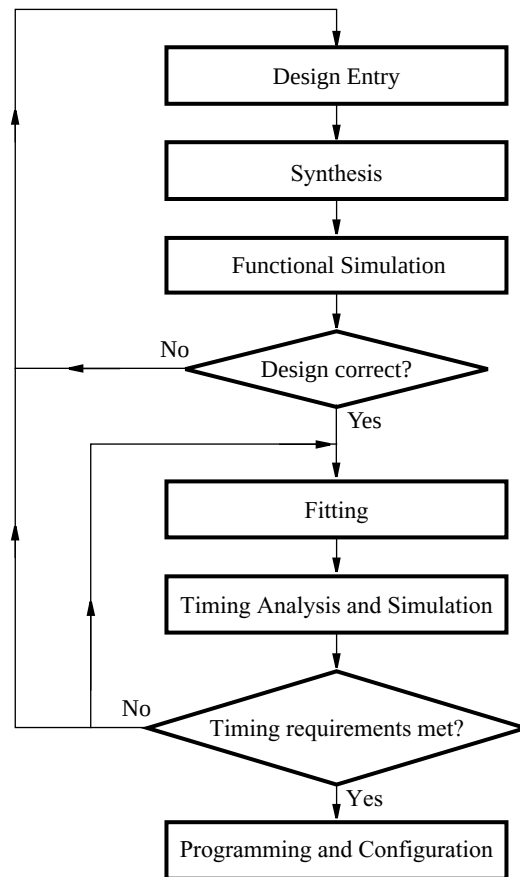


Figure 1. Typical CAD flow.

The CAD flow involves the following steps:

- **Design Entry** – the desired circuit is specified either by means of a schematic diagram, or by using a hardware description language, such as Verilog or VHDL
- **Synthesis** – the entered design is synthesized into a circuit that consists of the logic elements (LEs) provided in the FPGA chip
- **Functional Simulation** – the synthesized circuit is tested to verify its functional correctness; this simulation does not take into account any timing issues

- **Fitting** – the CAD Fitter tool determines the placement of the LEs defined in the netlist into the LEs in an actual FPGA chip; it also chooses routing wires in the chip to make the required connections between specific LEs
- **Timing Analysis** – propagation delays along the various paths in the fitted circuit are analyzed to provide an indication of the expected performance of the circuit
- **Timing Simulation** – the fitted circuit is tested to verify both its functional correctness and timing
- **Programming and Configuration** – the designed circuit is implemented in a physical FPGA chip by programming the configuration switches that configure the LEs and establish the required wiring connections

This tutorial introduces the basic features of the Quartus II software. It shows how the software can be used to design and implement a circuit specified by using the Verilog hardware description language. It makes use of the graphical user interface to invoke the Quartus II commands. Doing this tutorial, the reader will learn about:

- Creating a project
- Design entry using Verilog code
- Synthesizing a circuit specified in Verilog code
- Fitting a synthesized circuit into an Altera FPGA
- Assigning the circuit inputs and outputs to specific pins on the FPGA
- Simulating the designed circuit
- Programming and configuring the FPGA chip on Altera's DE2 board

1 Getting Started

Each logic circuit, or subcircuit, being designed with Quartus II software is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory (folder) in the file system. To begin a new logic circuit design, the first step is to create a directory to hold its files. To hold the design files for this tutorial, we will use a directory *introtutorial*. The running example for this tutorial is a simple circuit for two-way light control.

Start the Quartus II software. You should see a display similar to the one in Figure 2. This display consists of several windows that provide access to all the features of Quartus II software, which the user selects with the computer mouse. Most of the commands provided by Quartus II software can be accessed by using a set of menus that are located below the title bar. For example, in Figure 2 clicking the left mouse button on the menu named File opens the menu shown in Figure 3. Clicking the left mouse button on the entry Exit exits from Quartus II software. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to press. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly.

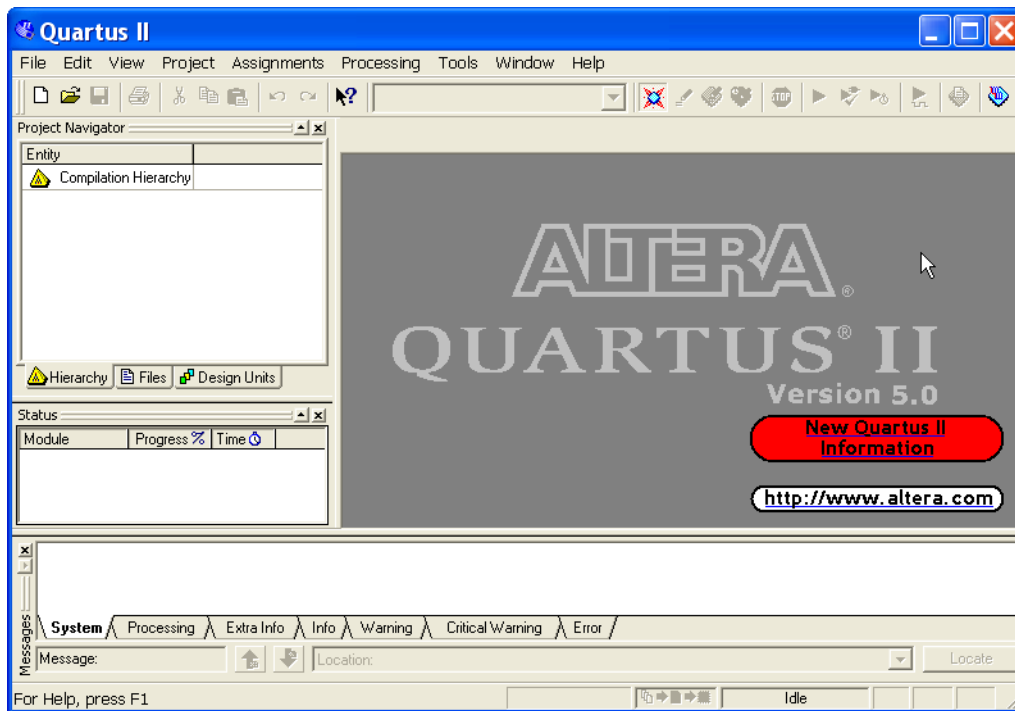


Figure 2. The main Quartus II display.

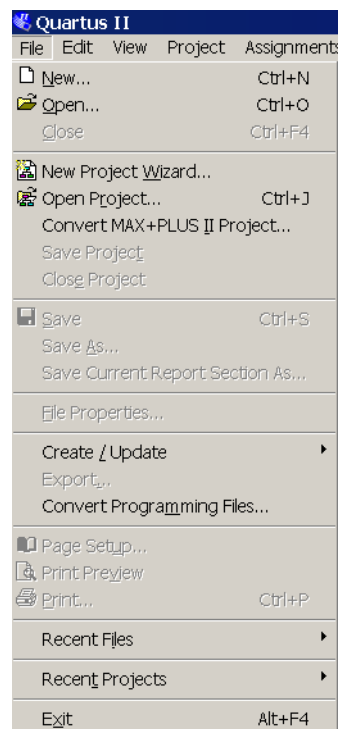


Figure 3. An example of the File menu.

For some commands it is necessary to access two or more menus in sequence. We use the convention **Menu1 > Menu2 > Item** to indicate that to select the desired command the user should first click the left mouse button on **Menu1**, then within this menu click on **Menu2**, and then within **Menu2** click on **Item**. For example, **File > Exit** uses the mouse to exit from the system. Many commands can be invoked by clicking on an icon displayed in one of the toolbars. To see the command associated with an icon, position the mouse over the icon and a tooltip will appear that displays the command name.

1.1 Quartus II Online Help

Quartus II software provides comprehensive online documentation that answers many of the questions that may arise when using the software. The documentation is accessed from the menu in the **Help** window. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the **Help** menu. For instance, selecting **Help > How to Use Help** gives an indication of what type of help is provided.

The user can quickly search through the Help topics by selecting **Help > Search**, which opens a dialog box into which key words can be entered. Another method, context-sensitive help, is provided for quickly finding documentation for specific topics. While using most applications, pressing the **F1** function key on the keyboard opens a Help display that shows the commands available for the application.

2 Starting a New Project

To start working on a new design we first have to define a new *design project*. Quartus II software makes the designer's task easy by providing support in the form of a *wizard*. Create a new project as follows:

1. Select **File > New Project Wizard** to reach the window in Figure 4, which indicates the capability of this wizard. You can skip this window in subsequent projects by checking the box **Don't show me this introduction again**. Press **Next** to get the window shown in Figure 5.

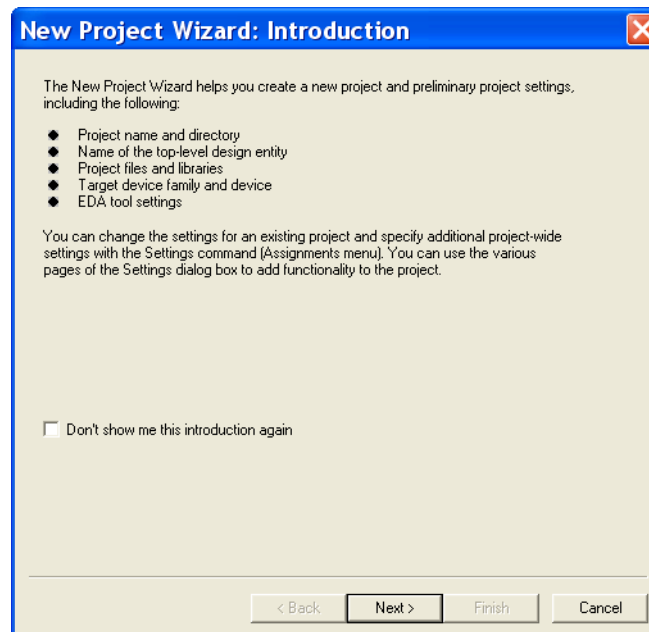


Figure 4. Tasks performed by the wizard.

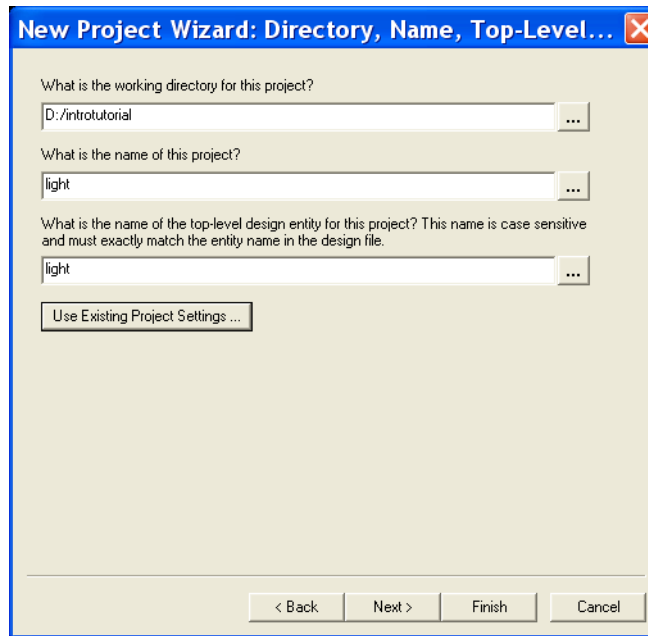


Figure 5. Creation of a new project.

2. Set the working directory to be *introtutorial*; of course, you can use some other directory name of your choice if you prefer. The project must have a name, which is usually the same as the top-level design entity that will be included in the project. Choose *light* as the name for both the project and the top-level entity, as shown in Figure 5. Press **Next**. Since we have not yet created the directory *introtutorial*, Quartus II software displays the pop-up box in Figure 6 asking if it should create the desired directory. Click **Yes**, which leads to the window in Figure 7.

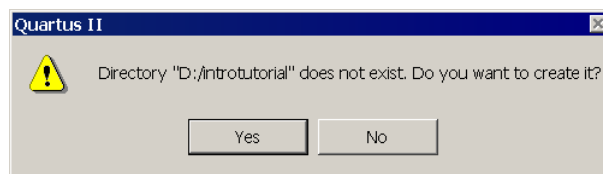


Figure 6. Quartus II software can create a new directory for the project.

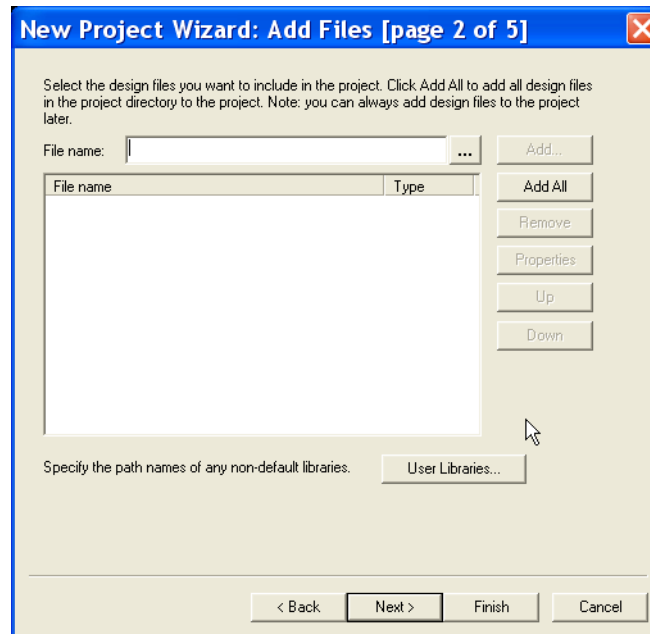


Figure 7. The wizard can include user-specified design files.

3. The wizard makes it easy to specify which existing files (if any) should be included in the project. Assuming that we do not have any existing files, click **Next**, which leads to the window in Figure 8.

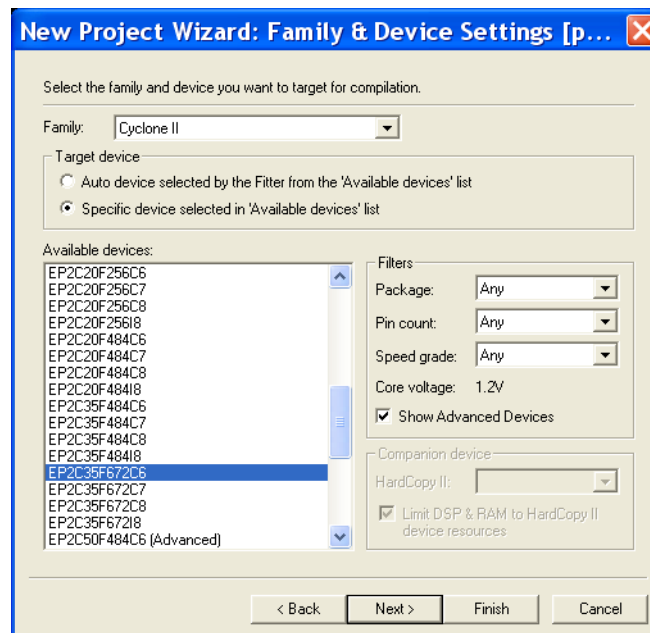


Figure 8. Choose the device family and a specific device.

4. We have to specify the type of device in which the designed circuit will be implemented. Choose Cyclone™ II as the target device family. We can let Quartus II software select a specific device in the family, or we can choose the device explicitly. We will take the latter approach. From the list of available devices, choose the device called EP2C35F672C6 which is the FPGA used on Altera's DE2 board. Press Next, which opens the window in Figure 9.

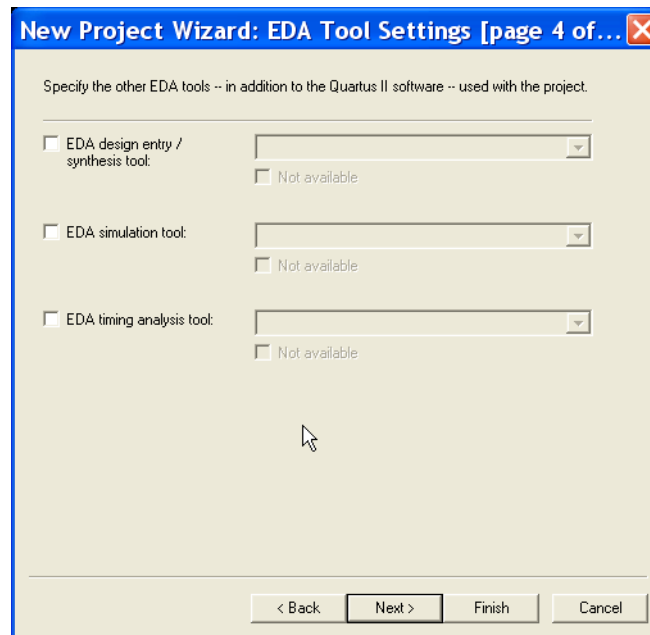


Figure 9. Other EDA tools can be specified.

5. The user can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is *EDA tools*, where the acronym stands for Electronic Design Automation. This term is used in Quartus II messages that refer to third-party tools, which are the tools developed and marketed by companies other than Altera. Since we will rely solely on Quartus II tools, we will not choose any other tools. Press Next.
6. A summary of the chosen settings appears in the screen shown in Figure 10. Press Finish, which returns to the main Quartus II window, but with *light* specified as the new project, in the display title bar, as indicated in Figure 11.

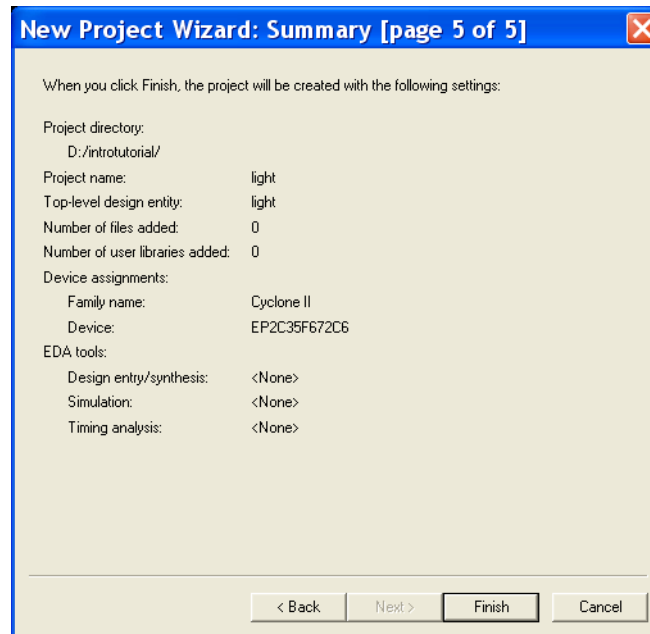


Figure 10. Summary of the project settings.

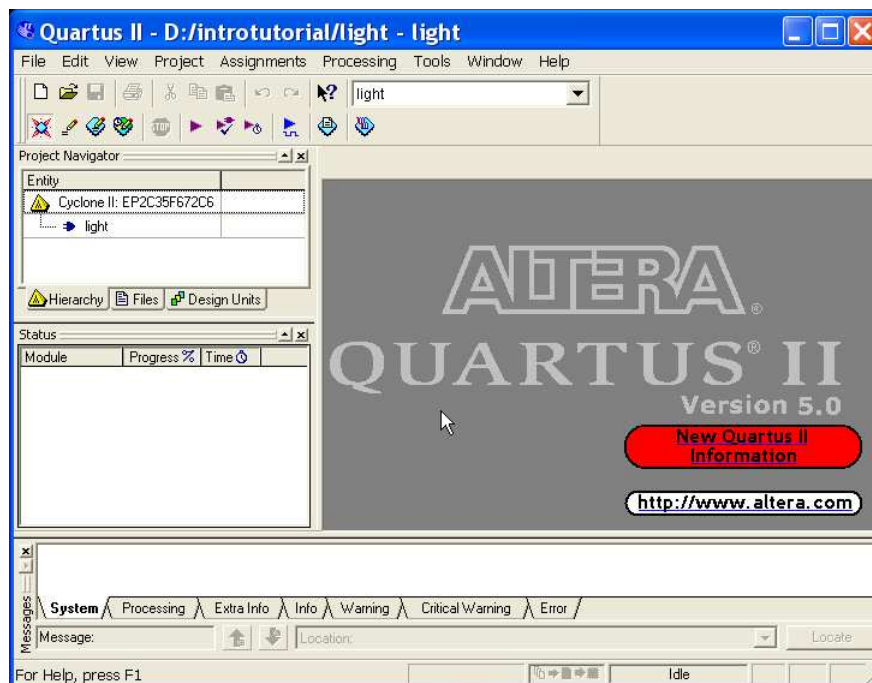


Figure 11. The Quartus II display for the created project.

3 Design Entry Using Verilog Code

As a design example, we will use the two-way light controller circuit shown in Figure 12. The circuit can be used to control a single light from either of the two switches, x_1 and x_2 , where a closed switch corresponds to the logic value 1. The truth table for the circuit is also given in the figure. Note that this is just the Exclusive-OR function of the inputs x_1 and x_2 , but we will specify it using the gates shown.

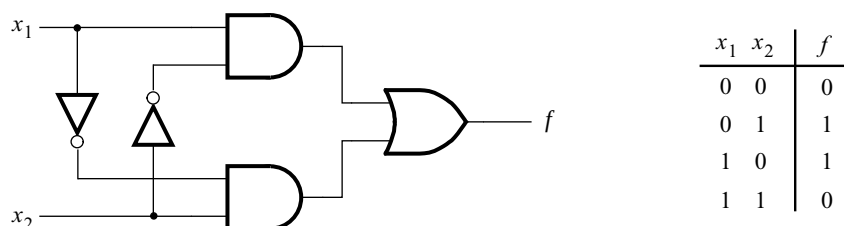


Figure 12. The light controller circuit.

The required circuit is described by the Verilog code in Figure 13. Note that the Verilog module is called *light* to match the name given in Figure 5, which was specified when the project was created. This code can be typed into a file by using any text editor that stores ASCII files, or by using the Quartus II text editing facilities. While the file can be given any name, it is a common designers' practice to use the same name as the name of the top-level Verilog module. The file name must include the extension *v*, which indicates a Verilog file. So, we will use the name *light.v*.

```
module light (x1, x2, f);  
    input x1, x2;  
    output f;  
    assign f = (x1 & ~x2) | (~x1 & x2);  
endmodule
```

Figure 13. Verilog code for the circuit in Figure 12.

3.1 Using the Quartus II Text Editor

This section shows how to use the Quartus II Text Editor. You can skip this section if you prefer to use some other text editor to create the Verilog source code file, which we will name *light.v*.

Select File > New to get the window in Figure 14, choose Verilog HDL File, and click OK. This opens the Text Editor window. The first step is to specify a name for the file that will be created. Select File > Save As to open the pop-up box depicted in Figure 15. In the box labeled Save as type choose Verilog HDL File. In the box labeled File name type *light*. Put a checkmark in the box Add file to current project. Click Save, which puts the file into the directory *introtutorial* and leads to the Text Editor window shown in Figure 16. Maximize the Text Editor window and enter the Verilog code in Figure 13 into it. Save the file by typing File > Save, or by typing the shortcut Ctrl-s.

Most of the commands available in the Text Editor are self-explanatory. Text is entered at the *insertion point*, which is indicated by a thin vertical line. The insertion point can be moved either by using the keyboard arrow keys or by using the mouse. Two features of the Text Editor are especially convenient for typing Verilog code. First, the editor can display different types of Verilog statements in different colors, which is the default choice. Second, the editor can automatically indent the text on a new line so that it matches the previous line. Such options can be controlled by the settings in Tools > Options > Text Editor.

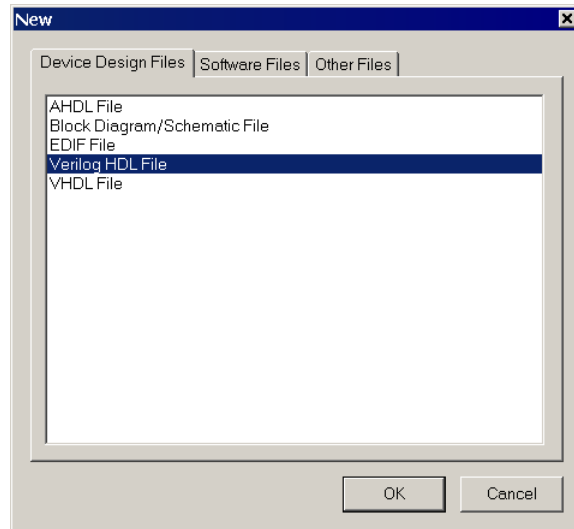


Figure 14. Choose to prepare a Verilog file.

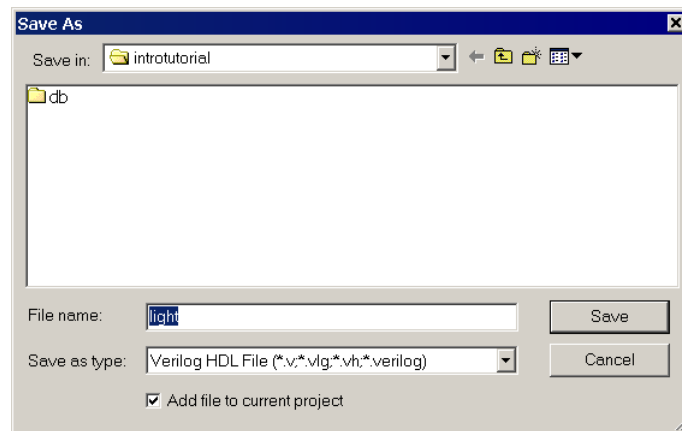


Figure 15. Name the file.



Figure 16. Text Editor window.

3.1.1 Using Verilog Templates

The syntax of Verilog code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of *Verilog templates*. The templates provide examples of various types of Verilog statements, such as a **module** declaration, an **always** block, and assignment statements. It is worthwhile to browse through the templates by selecting Edit > Insert Template > Verilog HDL to become familiar with this resource.

3.2 Adding Design Files to a Project

As we indicated when discussing Figure 7, you can tell Quartus II software which design files it should use as part of the current project. To see the list of files already included in the *light* project, select Assignments > Settings, which leads to the window in Figure 17. As indicated on the left side of the figure, click on the item Files. An alternative way of making this selection is to choose Project > Add/Remove Files in Project.

If you used the Quartus II Text Editor to create the file and checked the box labeled Add file to current project, as described in Section 3.1, then the *light.v* file is already a part of the project and will be listed in the window in Figure 17. Otherwise, the file must be added to the project. So, if you did not use the Quartus II Text Editor, then place a copy of the file *light.v*, which you created using some other text editor, into the directory *introtutorial*. To add this file to the project, click on the File name: button in Figure 17 to get the pop-up window in Figure 18. Select the *light.v* file and click Open. The selected file is now indicated in the Files window of Figure 17. Click OK to include the *light.v* file in the project. We should mention that in many cases the Quartus II software is able to automatically find the right files to use for each entity referenced in Verilog code, even if the file has not been explicitly added to the project. However, for complex projects that involve many files it is a good design practice to specifically add the needed files to the project, as described above.

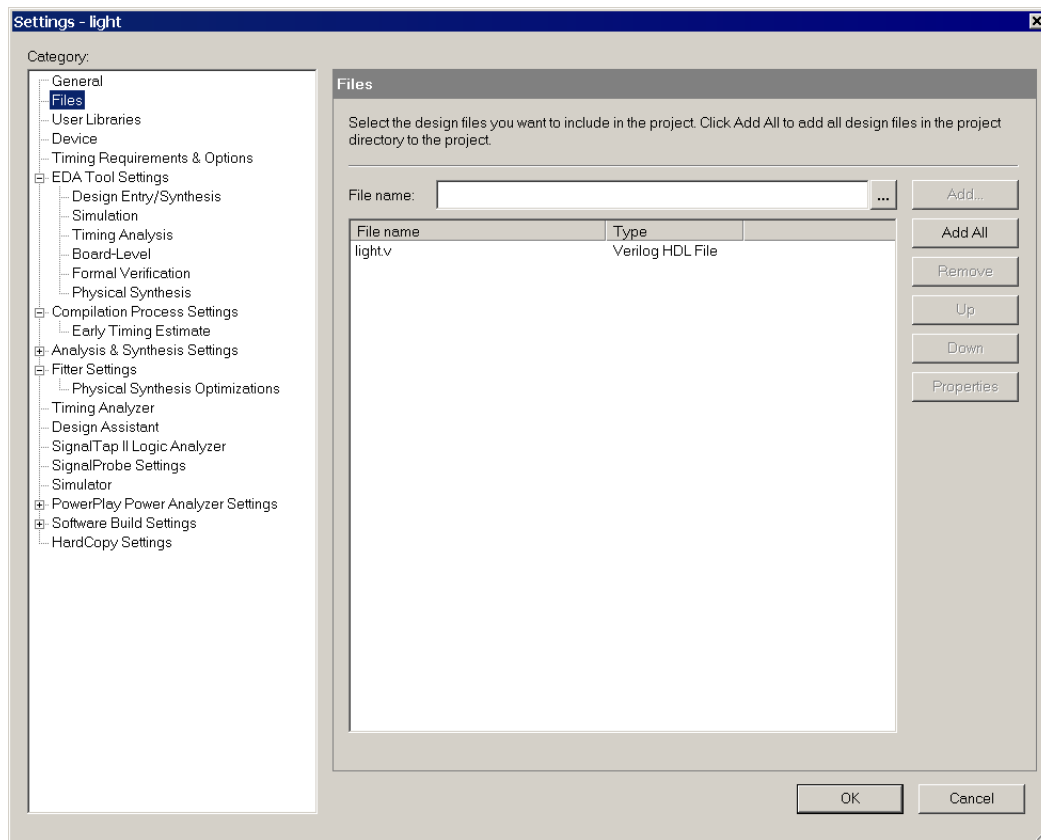


Figure 17. Settings window.

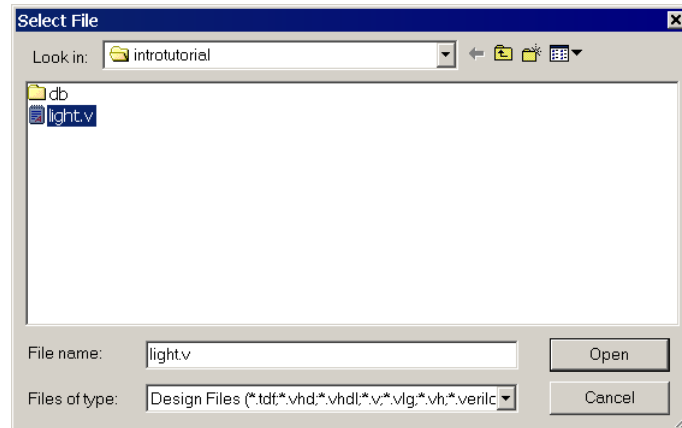


Figure 18. Select the file.

4 Compiling the Designed Circuit

The Verilog code in the file *light.v* is processed by several Quartus II tools that analyze the code, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the *Compiler*.

Run the Compiler by selecting **Processing > Start Compilation**, or by clicking on the toolbar icon that looks like a purple triangle. As the compilation moves through various stages, its progress is reported in a window on the left side of the Quartus II display. Successful (or unsuccessful) compilation is indicated in a pop-up box. Acknowledge it by clicking OK, which leads to the Quartus II display in Figure 19. In the message window, at the bottom of the figure, various messages are displayed. In case of errors, there will be appropriate messages given.

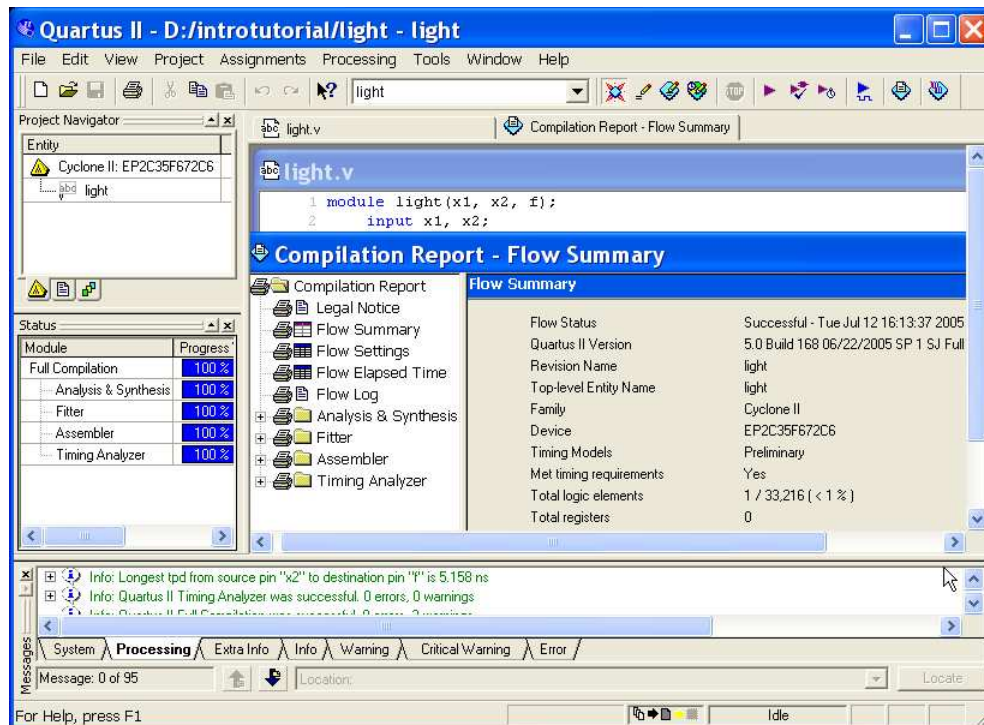


Figure 19. Display after a successful compilation.

When the compilation is finished, a compilation report is produced. A window showing this report is opened automatically, as seen in Figure 19. The window can be resized, maximized, or closed in the normal way, and it can be opened at any time either by selecting **Processing > Compilation Report** or by clicking on the icon . The report includes a number of sections listed on the left side of its window. Figure 19 displays the Compiler Flow Summary section, which indicates that only one logic element and three pins are needed to implement this tiny circuit on the selected FPGA chip. Another section is shown in Figure 20. It is reached by selecting **Analysis & Synthesis > Equations** on the left side of the compilation report. Here we see the logic expressions produced by the Compiler when synthesizing the designed circuit. Observe that f is the output derived as

$$f = x2 \$ x1$$

where the \$ sign is used to represent the Exclusive-OR operation. Obviously, the Compiler recognized that the logic expression in our design file is equivalent to this expression.

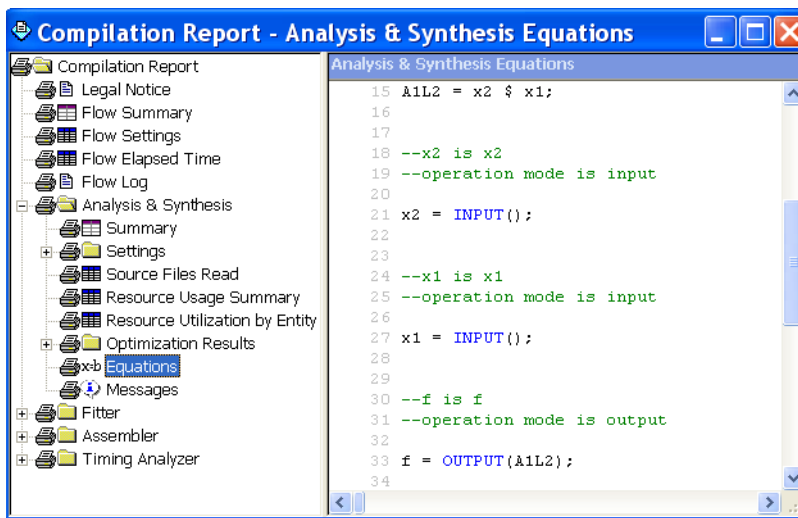


Figure 20. Compilation report showing the synthesized equations.

4.1 Errors

Quartus II software displays messages produced during compilation in the Messages window. If the Verilog design file is correct, one of the messages will state that the compilation was successful and that there are no errors.

If the Compiler does not report zero errors, then there is at least one mistake in the Verilog code. In this case a message corresponding to each error found will be displayed in the Messages window. Double-clicking on an error message will highlight the offending statement in the Verilog code in the Text Editor window. Similarly, the Compiler may display some warning messages. Their details can be explored in the same way as in the case of error messages. The user can obtain more information about a specific error or warning message by selecting the message and pressing the F1 function key.

To see the effect of an error, open the file *light.v*. Remove the semicolon in the **assign** statement, illustrating a typographical error that is easily made. Compile the erroneous design file by clicking on the  icon. A pop-up box will ask if the changes made to the *light.v* file should be saved; click **Yes**. After trying to compile the circuit, Quartus II software will display a pop-up box indicating that the compilation was not successful. Acknowledge it by clicking **OK**. The compilation report summary, given in Figure 21, now confirms the failed result. Expand the **Analysis & Synthesis** part of the report and then select **Messages** to have the messages displayed as shown in Figure 22. Double-click on the first error message. Quartus II software responds by opening the *light.v* file and highlighting the statement which is affected by the error, as shown in Figure 23. Correct the error and recompile

the design.

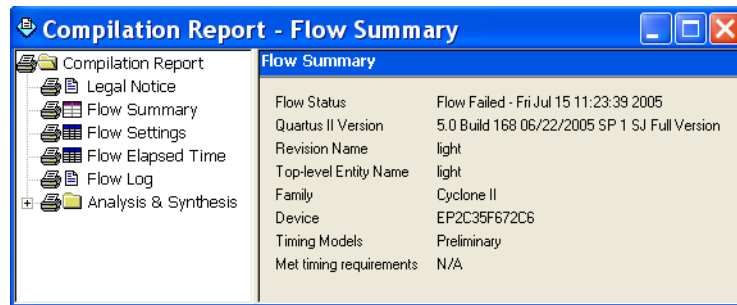


Figure 21. Compilation report for the failed design.

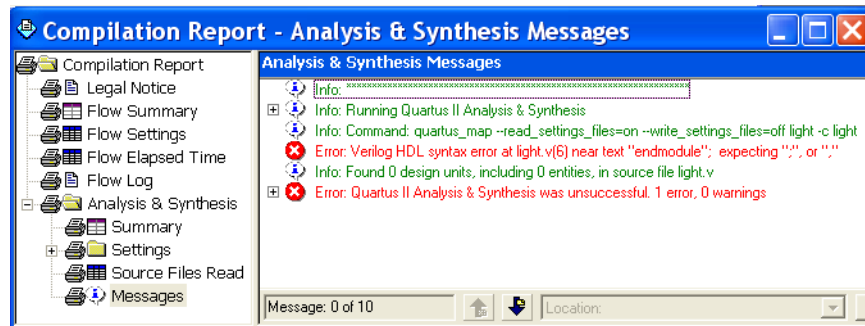


Figure 22. Error messages.

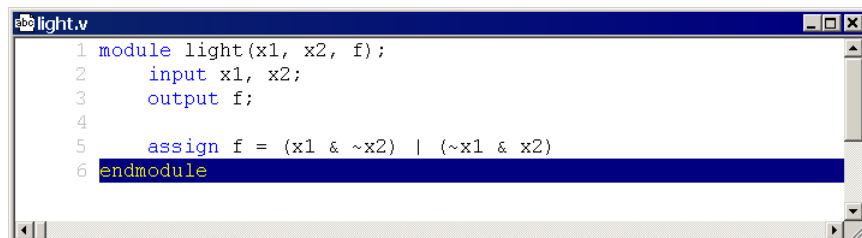


Figure 23. Identifying the location of the error.

5 Pin Assignment

During the compilation above, the Quartus II Compiler was free to choose any pins on the selected FPGA to serve as inputs and outputs. However, the DE2 board has hardwired connections between the FPGA pins and the other components on the board. We will use two toggle switches, labeled SW_0 and SW_1 , to provide the external inputs, x_1 and x_2 , to our example circuit. These switches are connected to the FPGA pins N25 and N26, respectively. We will connect the output f to the green light-emitting diode labeled $LEDG_0$, which is hardwired to the FPGA pin AE22.

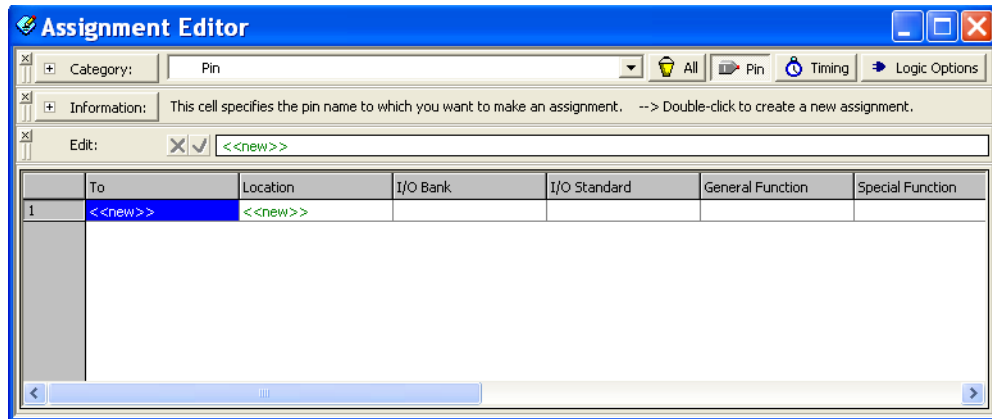


Figure 24. The Assignment Editor window.

Pin assignments are made by using the Assignment Editor. Select **Assignments > Pins** to reach the window in Figure 24. Under **Category** select **Pin**. Double-click on the entry **<<new>>** which is highlighted in blue in the column labeled **To**. The drop-down menu in Figure 25 will appear. Click on *x1* as the first pin to be assigned; this will enter *x1* in the displayed table. Follow this by double-clicking on the box to the right of this new *x1* entry, in the column labeled **Location**. Now, the drop-down menu in Figure 26 appears. Scroll down and select **PIN_N25**. Instead of scrolling down the menu to find the desired pin, you can just type the name of the pin (N25) in the **Location** box. Use the same procedure to assign input *x2* to pin N26 and output *f* to pin AE22, which results in the image in Figure 27. To save the assignments made, choose **File > Save**. You can also simply close the Assignment Editor window, in which case a pop-up box will ask if you want to save the changes to assignments; click **Yes**. Recompile the circuit, so that it will be compiled with the correct pin assignments.

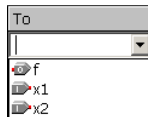


Figure 25. The drop-down menu displays the input and output names.

Location	I/O Bank	I/O Standard	General Function	Special Fun
<<new>>		LVTTL		
PIN_N1	I/O Bank 2	Dedicated Clock	CLK1, LVDSCLK0n, Input	
PIN_N2	I/O Bank 2	Dedicated Clock	CLK0, LVDSCLK0p, Input	
PIN_N9	I/O Bank 2	Row I/O	LVDS31p	
PIN_N18	I/O Bank 5	Row I/O	LVDS110p	
PIN_N20	I/O Bank 5	Row I/O	LVDS124p	
PIN_N23	I/O Bank 5	Row I/O	LVDS126p, DPCLK7/DQS0R/CQ1R	
PIN_N24	I/O Bank 5	Row I/O	LVDS126n	
PIN_N25	I/O Bank 5	Dedicated Clock	CLK4, LVDSCLK2p, Input	
PIN_N26	I/O Bank 5	Dedicated Clock	CLK5, LVDSCLK2n, Input	
PIN_P1	I/O Bank 1	Dedicated Clock	CLK3, LVDSCLK1n, Input	
PIN_P2	I/O Bank 1	Dedicated Clock	CLK2, LVDSCLK1p, Input	
PIN_P3	I/O Bank 1	Row I/O	LVDS26p, DPCLK1/DQS1L/CQ1L#	
PIN_P4	I/O Bank 1	Row I/O	LVDS26n	
PIN_P6	I/O Bank 1	Row I/O	LVDS22n	
PIN_P7	I/O Bank 1	Row I/O	LVDS22p	
PIN_P9	I/O Bank 2	Row I/O	LVDS31n	
PIN_P17	I/O Bank 6	Row I/O	LVDS130n	
PIN_P18	I/O Bank 5	Row I/O	LVDS110n	
PIN_P23	I/O Bank 6	Row I/O	LVDS127p, DPCLK6/DQS1R/CQ1R#	
PIN_P24	I/O Bank 6	Row I/O	LVDS127n	

Figure 26. The available pins.

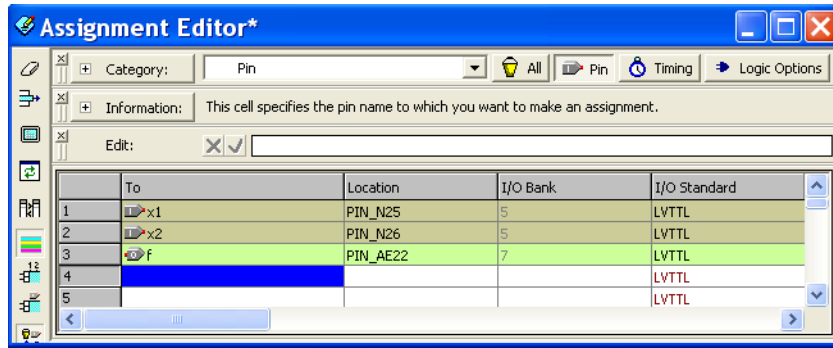


Figure 27. The complete assignment.

The DE2 board has fixed pin assignments. Having finished one design, the user will want to use the same pin assignment for subsequent designs. Going through the procedure described above becomes tedious if there are many pins used in the design. A useful Quartus II feature allows the user to both export and import the pin assignments from a special file format, rather than creating them manually using the Assignment Editor. A simple file format that can be used for this purpose is the *comma separated value (CSV)* format, which is a common text file format that contains comma-delimited values. This file format is often used in conjunction with the Microsoft Excel spreadsheet program, but the file can also be created by hand using any plain ASCII text editor. The format for the file for our simple project is

```
To, Location
x1, PIN_N25
x2, PIN_N26
f, PIN_AE22
```

By adding lines to the file, any number of pin assignments can be created. Such csv files can be imported into any design project.

If you created a pin assignment for a particular project, you can export it for use in a different project. To see how this is done, open again the Assignment Editor to reach the window in Figure 27. Now, select **File > Export** which leads to the window in Figure 28. Here, the file *light.csv* is available for export. Click on **Export**. If you now look in the directory *introtutorial*, you will see that the file *light.csv* has been created.

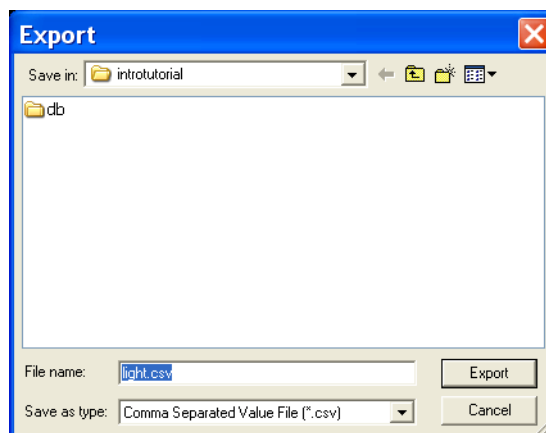


Figure 28. Exporting the pin assignment.

You can import a pin assignment by choosing **Assignments > Import Assignments**. This opens the dialogue in Figure 29 to select the file to import. Type the name of the file, including the csv extension and the full path to the directory that holds the file, in the File Name box and press OK. Of course, you can also browse to find the desired file.

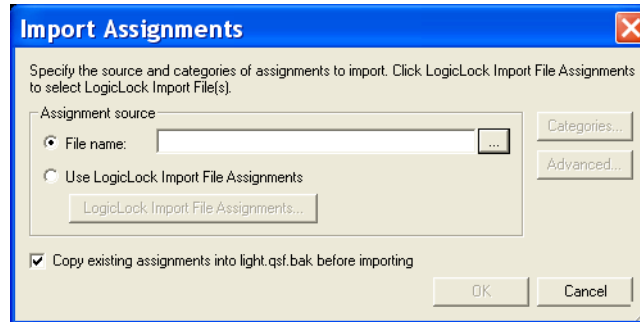


Figure 29. Importing the pin assignment.

For convenience when using large designs, all relevant pin assignments for the DE2 board are given in the file called *DE2_pin_assignments.csv* in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages. This file uses the names found in the *DE2 User Manual*. If we wanted to make the pin assignments for our example circuit by importing this file, then we would have to use the same names in our Verilog design file; namely, *SW[0]*, *SW[1]* and *LEDG[0]* for *x1*, *x2* and *f*, respectively. Since these signals are specified in the *DE2_pin_assignments.csv* file as elements of vectors *SW* and *LEDG*, we must refer to them in the same way in the Verilog design file. For example, in the *DE2_pin_assignments.csv* file the 18 toggle switches are called *SW[17]* to *SW[0]*. In Verilog code, they can also be referred to as a vector *SW[17:0]*.

6 Simulating the Designed Circuit

Before implementing the designed circuit in the FPGA chip on the DE2 board, it is prudent to simulate it to ascertain its correctness. Quartus II software includes a simulation tool that can be used to simulate the behavior of a designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. It is also necessary to specify which outputs, as well as possible internal points in the circuit, the designer wishes to observe. The simulator applies the test vectors to a model of the implemented circuit and determines the expected response. We will use the Quartus II Waveform Editor to draw the test vectors, as follows:

1. Open the Waveform Editor window by selecting **File > New**, which gives the window shown in Figure 30. Click on the **Other Files** tab to reach the window displayed in Figure 31. Choose **Vector Waveform File** and click OK.
2. The Waveform Editor window is depicted in Figure 32. Save the file under the name *light.vwf*; note that this changes the name in the displayed window. Set the desired simulation to run from 0 to 200 ns by selecting **Edit > End Time** and entering 200 ns in the dialog box that pops up. Selecting **View > Fit in Window** displays the entire simulation range of 0 to 200 ns in the window, as shown in Figure 33. You may wish to resize the window to its maximum size.

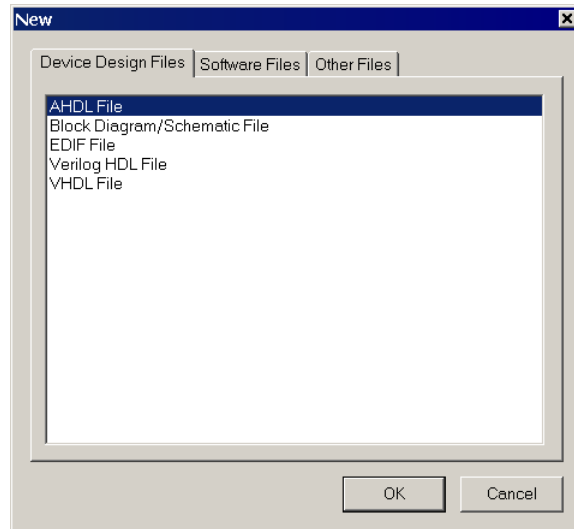


Figure 30. Need to prepare a new file.

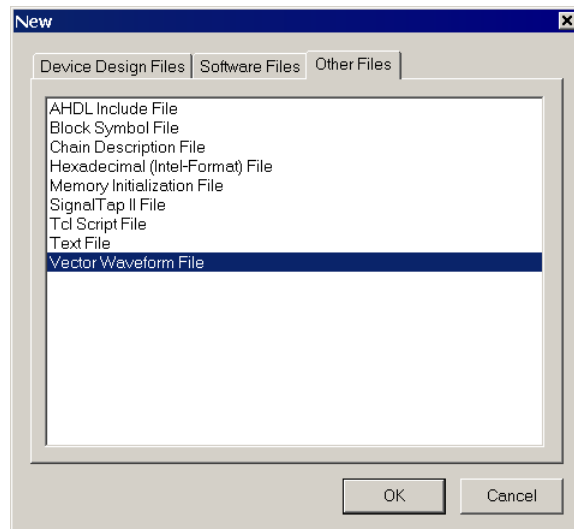


Figure 31. Choose to prepare a test-vector file.

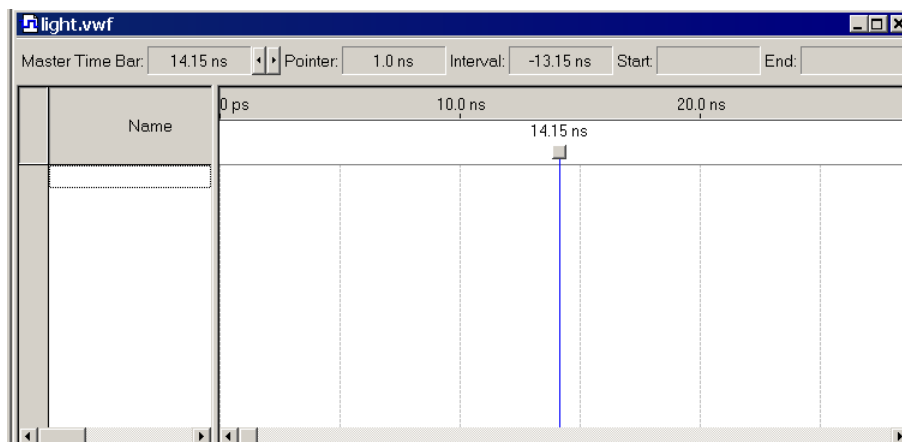


Figure 32. The Waveform Editor window.

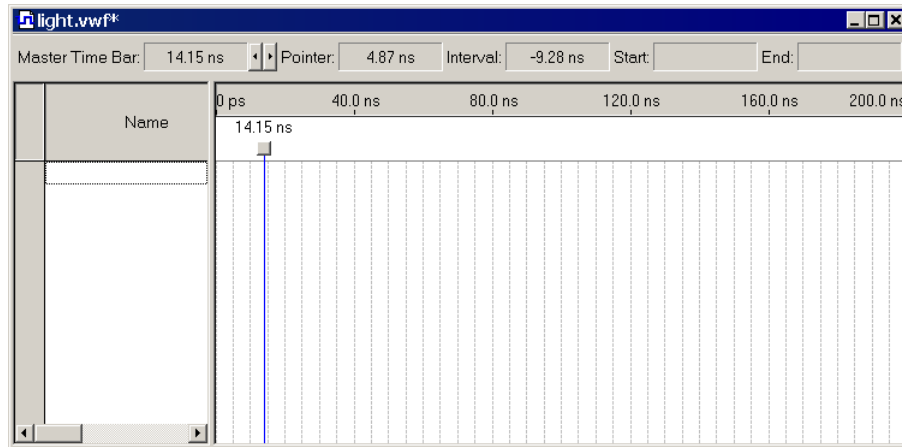


Figure 33. The augmented Waveform Editor window.

- Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert Node or Bus** to open the window in Figure 34. It is possible to type the name of a signal (pin) into the Name box, but it is easier to click on the button labeled **Node Finder** to open the window in Figure 35. The Node Finder utility has a filter used to indicate what type of nodes are to be found. Since we are interested in input and output pins, set the filter to **Pins: all**. Click the **List** button to find the input and output nodes as indicated on the left side of the figure.

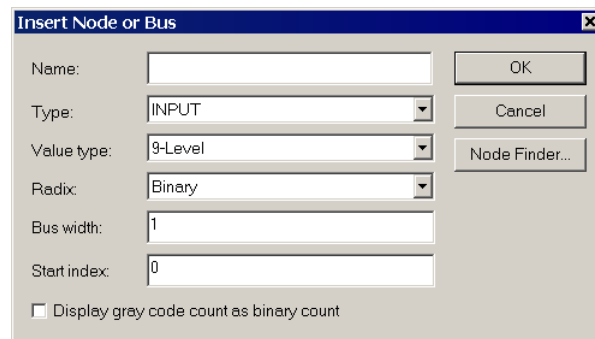


Figure 34. The Insert Node or Bus dialogue.

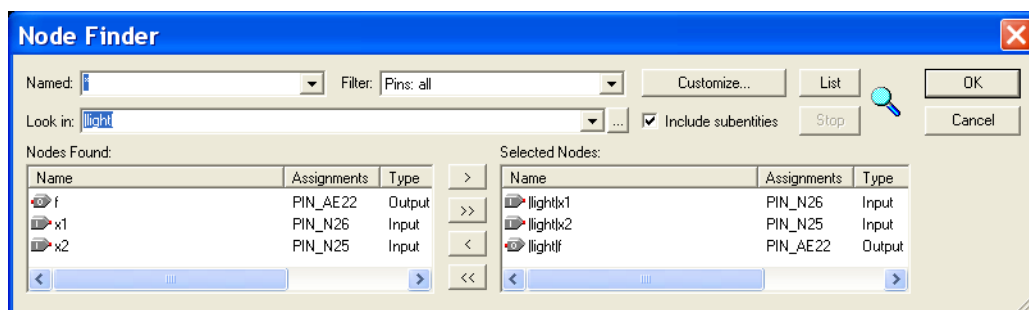


Figure 35. Selecting nodes to insert into the Waveform Editor.

Click on the $x1$ signal in the Nodes Found box in Figure 35, and then click the > sign to add it to the Selected Nodes box on the right side of the figure. Do the same for $x2$ and f . Click OK to close the Node Finder window, and then click OK in the window of Figure 34. This leaves a fully displayed Waveform Editor window, as shown in Figure 36. If you did not select the nodes in the same order as displayed in Figure 36, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click on the node name (in the Name column) and release the mouse button. The waveform is now highlighted to show the selection. Click again on the waveform and drag it up or down in the Waveform Editor.

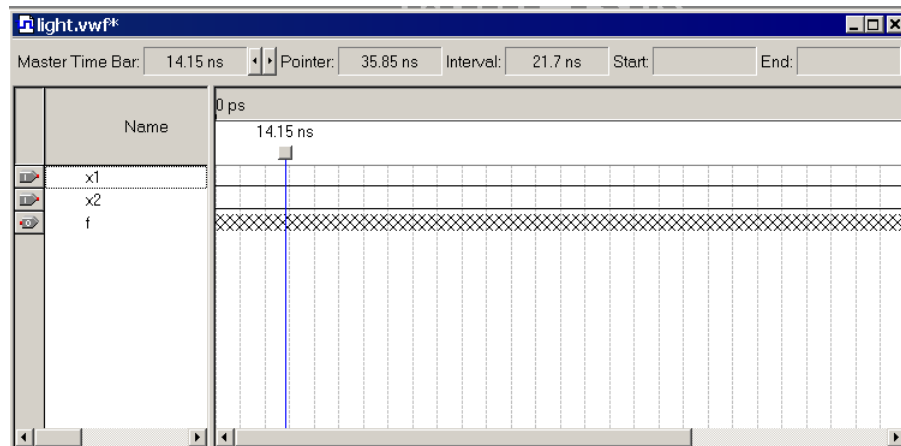


Figure 36. The nodes needed for simulation.

4. We will now specify the logic values to be used for the input signals $x1$ and $x2$ during simulation. The logic values at the output f will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing View > Snap to Grid). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This reference line is used in analyzing the timing of a circuit; move it to the $time = 0$ position. The waveforms can be drawn using the Selection Tool, which is activated by selecting the icon  in the toolbar, or the Waveform Editing Tool, which is activated by the icon .

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. In a large circuit the number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations. However, for our tiny circuit we can simulate all four input valuations given in Figure 12. We will use four 50-ns time intervals to apply the four test vectors.

We can generate the desired input waveforms as follows. Click on the waveform name for the $x1$ node. Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for setting a selected signal to 0, 1, unknown (X), high impedance (Z), don't care (DC), inverting its existing value (INV), or defining a clock waveform. Each command can be activated by using the Edit > Value command, or via the toolbar for the Waveform Editor. The Edit menu can also be opened by right-clicking on a waveform name.

Set $x1$ to 0 in the time interval 0 to 100 ns, which is probably already set by default. Next, set $x1$ to 1 in the time interval 100 to 200 ns. Do this by pressing the mouse at the start of the interval and dragging it to its end, which highlights the selected interval, and choosing the logic value 1 in the toolbar. Make $x2 = 1$ from 50 to 100 ns and also from 150 to 200 ns, which corresponds to the truth table in Figure 12. This should

produce the image in Figure 37. Observe that the output f is displayed as having an unknown value at this time, which is indicated by a hashed pattern; its value will be determined during simulation. Save the file.

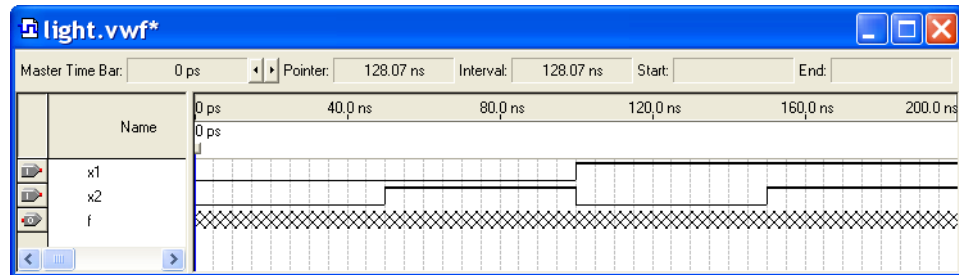


Figure 37. Setting of test values.

6.1 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and interconnection wires in the FPGA are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit.

6.1.1 Functional Simulation

To perform the functional simulation, select **Assignments > Settings** to open the Settings window. On the left side of this window click on **Simulator** to display the window in Figure 38, choose **Functional** as the simulation mode, and click OK. The Quartus II simulator takes the inputs and generates the outputs defined in the *light.vwf* file. Before running the functional simulation it is necessary to create the required netlist, which is done by selecting **Processing > Generate Functional Simulation Netlist**. A simulation run is started by **Processing > Start Simulation**, or by using the icon . At the end of the simulation, Quartus II software indicates its successful completion and displays a Simulation Report illustrated in Figure 39. If your report window does not show the entire simulation time range, click on the report window to select it and choose **View > Fit in Window**. Observe that the output f is as specified in the truth table of Figure 12.

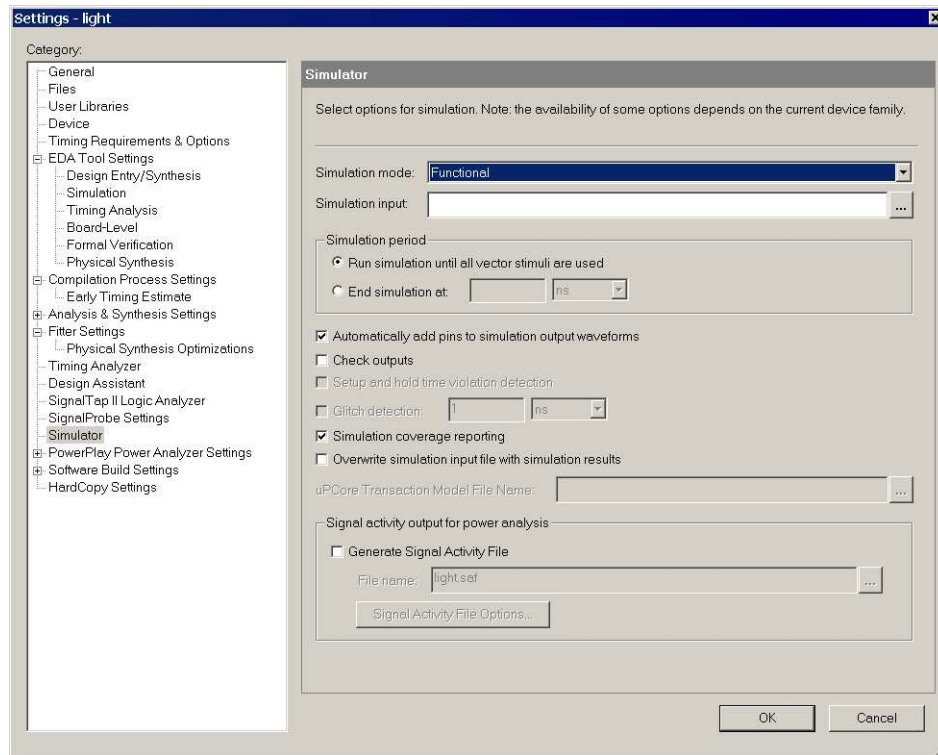


Figure 38. Specifying the simulation mode.

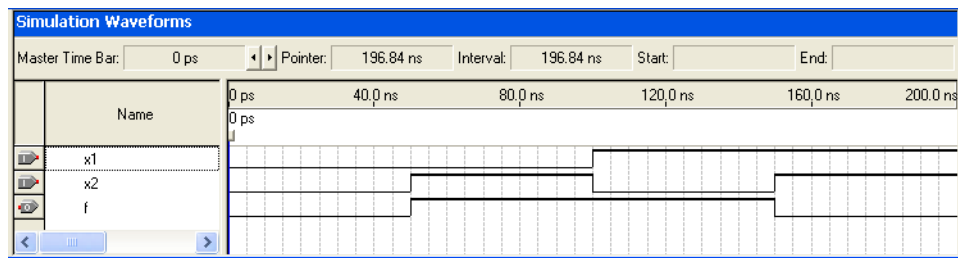


Figure 39. The result of functional simulation.

6.1.2 Timing Simulation

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how it will behave when it is actually implemented in the chosen FPGA device. Select **Assignments > Settings > Simulator** to get to the window in Figure 38, choose **Timing** as the simulation mode, and click **OK**. Run the simulator, which should produce the waveforms in Figure 40. Observe that there is a delay of about 6 ns in producing a change in the signal f from the time when the input signals, x_1 and x_2 , change their values. This delay is due to the propagation delays in the logic element and the wires in the FPGA device. You may also notice that a momentary change in the value of f , from 1 to 0 and back to 1, occurs at about 106-ns point in the simulation. This *glitch* is also due to the propagation delays in the FPGA device, because changes in x_1 and x_2 may not arrive at exactly the same time at the logic element that generates f .

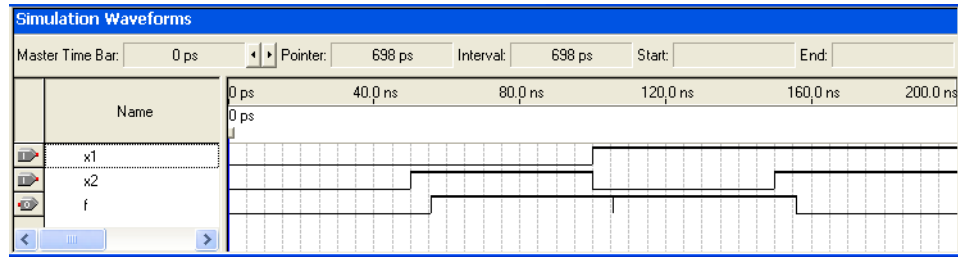


Figure 40. The result of timing simulation.

7 Programming and Configuring the FPGA Device

The FPGA device must be programmed and configured to implement the designed circuit. The required configuration file is generated by the Quartus II Compiler's Assembler module. Altera's DE2 board allows the configuration to be done in two different ways, known as JTAG and AS modes. The configuration data is transferred from the host computer (which runs the Quartus II software) to the board by means of a cable that connects a USB port on the host computer to the leftmost USB connector on the board. To use this connection, it is necessary to have the USB-Blaster driver installed. If this driver is not already installed, consult the tutorial *Getting Started with Altera's DE2 Board* for information about installing the driver. Before using the board, make sure that the USB cable is properly connected and turn on the power supply switch on the board.

In the JTAG mode, the configuration data is loaded directly into the FPGA device. The acronym JTAG stands for Joint Test Action Group. This group defined a simple way for testing digital circuits and loading data into them, which became an IEEE standard. If the FPGA is configured in this manner, it will retain its configuration as long as the power remains turned on. The configuration information is lost when the power is turned off. The second possibility is to use the Active Serial (AS) mode. In this case, a configuration device that includes some flash memory is used to store the configuration data. Quartus II software places the configuration data into the configuration device on the DE2 board. Then, this data is loaded into the FPGA upon power-up or reconfiguration. Thus, the FPGA need not be configured by the Quartus II software if the power is turned off and on. The choice between the two modes is made by the RUN/PROG switch on the DE2 board. The RUN position selects the JTAG mode, while the PROG position selects the AS mode.

7.1 JTAG Programming

The programming and configuration task is performed as follows. Flip the RUN/PROG switch into the RUN position. Select Tools > Programmer to reach the window in Figure 41. Here it is necessary to specify the programming hardware and the mode that should be used. If not already chosen by default, select JTAG in the Mode box. Also, if the USB-Blaster is not chosen by default, press the Hardware Setup... button and select the USB-Blaster in the window that pops up, as shown in Figure 42.

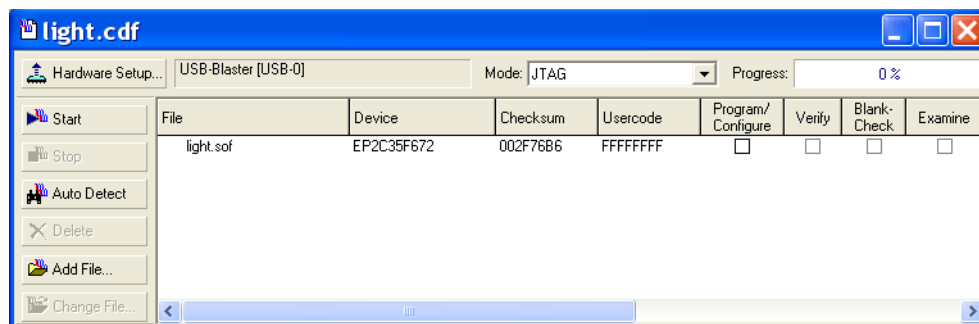


Figure 41. The Programmer window.

Observe that the configuration file *light.sof* is listed in the window in Figure 41. If the file is not already listed, then click **Add File** and select it. This is a binary file produced by the Compiler's Assembler module, which contains the data needed to configure the FPGA device. The extension *.sof* stands for SRAM Object File. Note also that the device selected is EP2C35F672, which is the FPGA device used on the DE2 board. Click on the **Program/Configure** check box, as shown in Figure 43.

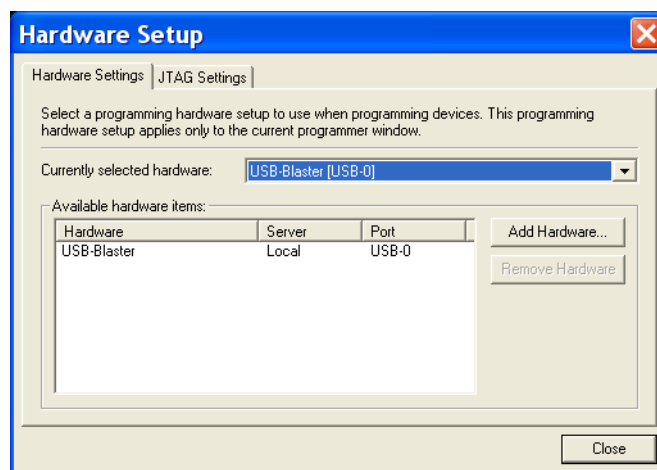


Figure 42. The Hardware Setup window.

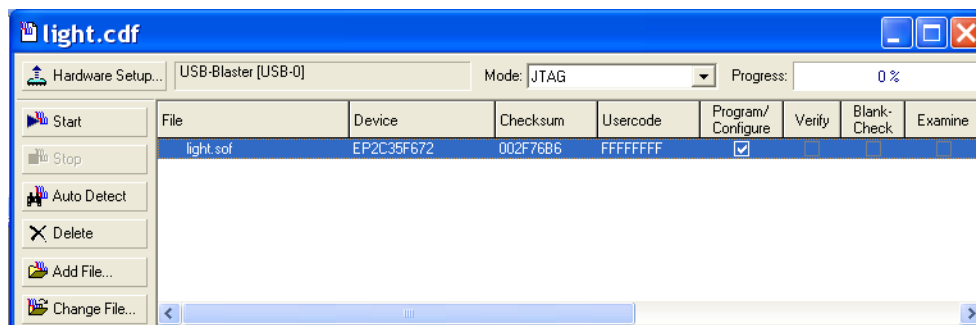


Figure 43. The updated Programmer window.

Now, press **Start** in the window in Figure 43. An LED on the board will light up when the configuration data has been downloaded successfully. If you see an error reported by Quartus II software indicating that programming failed, then check to ensure that the board is properly powered on.

7.2 Active Serial Mode Programming

In this case, the configuration data has to be loaded into the configuration device on the DE2 board, which is identified by the name EPCS16. To specify the required configuration device select **Assignments > Device**, which leads to the window in Figure 44. Click on the **Device & Pin Options** button to reach the window in Figure 45. Now, click on the **Configuration** tab to obtain the window in Figure 46. In the Configuration device box (which may be set to Auto) choose EPCS16 and click OK. Upon returning to the window in Figure 44, click OK. Recompile the designed circuit.

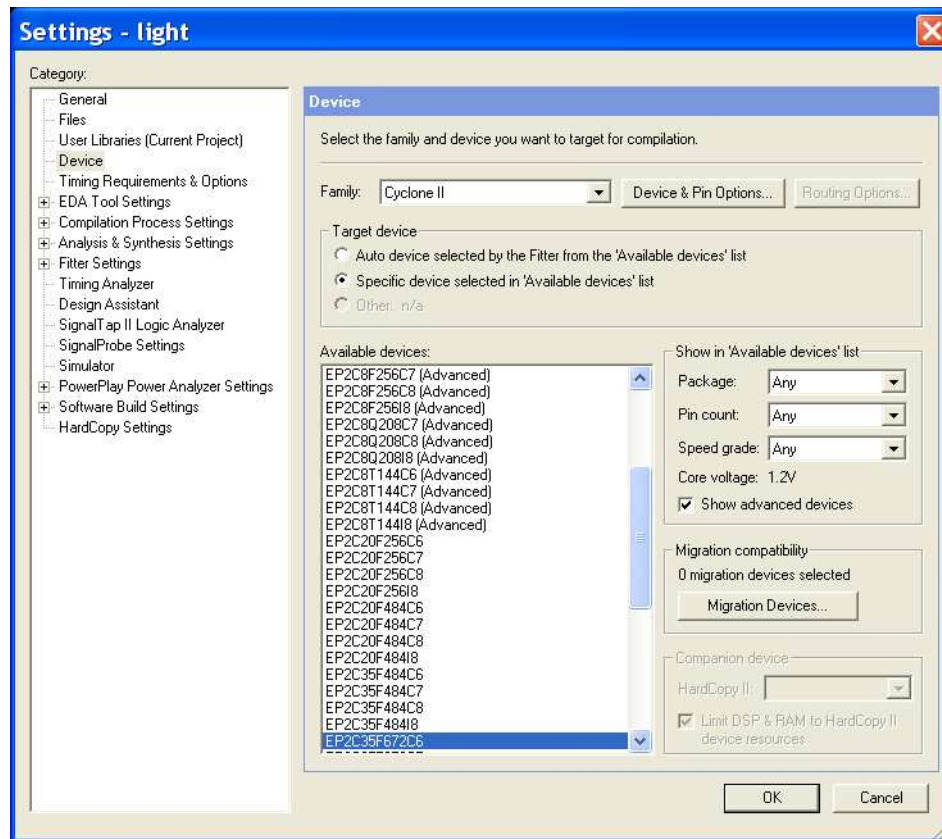


Figure 44. The Device Settings window.

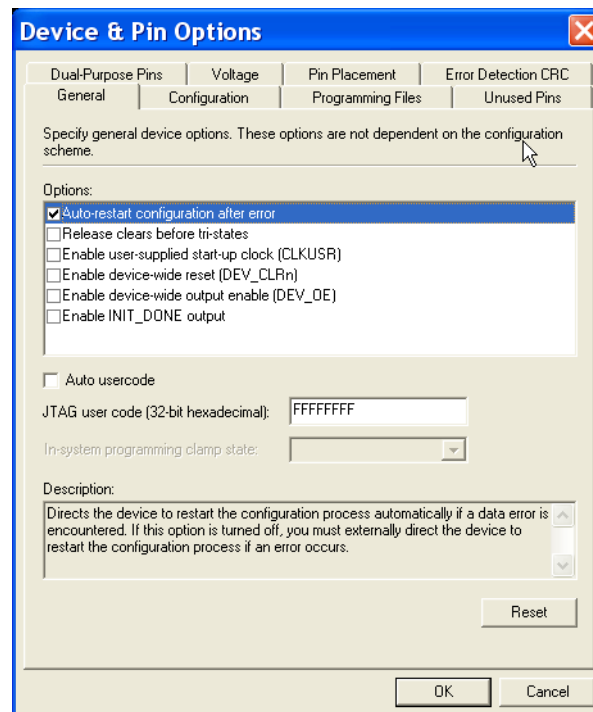


Figure 45. The Options window.

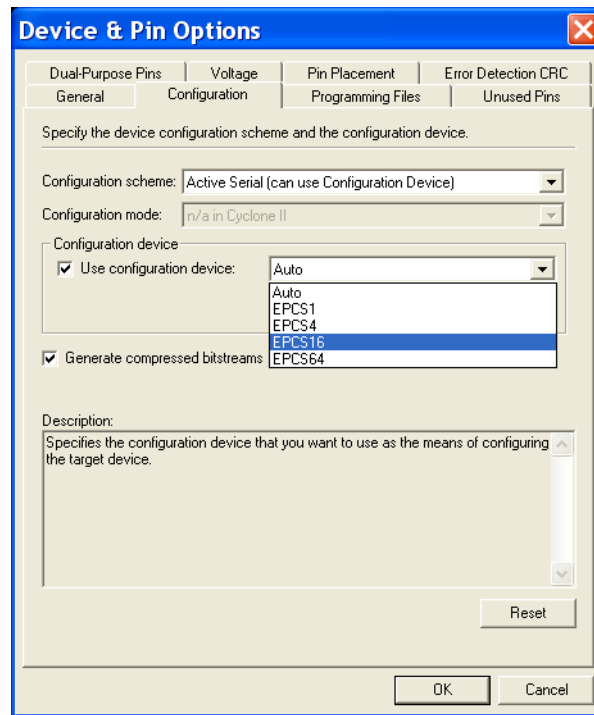


Figure 46. Specifying the configuration device.

The rest of the procedure is similar to the one described above for the JTAG mode. Select **Tools > Programmer** to reach the window in Figure 41. In the Mode box select **Active Serial Programming**. If you are changing the mode from the previously used JTAG mode, the pop-up box in Figure 47 will appear, asking if you want to clear all devices. Click **Yes**. Now, the Programmer window shown in Figure 48 will appear. Make sure that the Hardware Setup indicates the USB-Blaster. If the configuration file is not already listed in the window, press **Add File**. The pop-up box in Figure 49 will appear. Select the file *light.pof* in the directory *introtutorial* and click **Open**. As a result, the configuration file *light.pof* will be listed in the window. This is a binary file produced by the Compiler's Assembler module, which contains the data to be loaded into the EPCS16 configuration device. The extension *.pof* stands for Programmer Object File. Upon returning to the Programmer window, click on the **Program/Configure** check box, as shown in Figure 50.

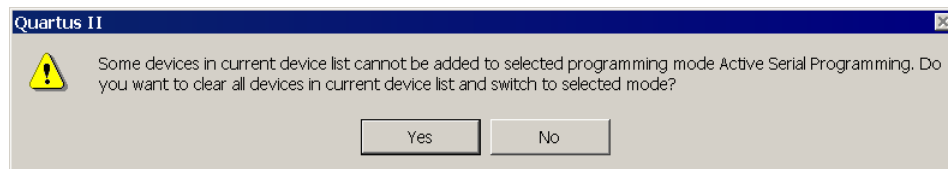


Figure 47. Clear the previously selected devices.

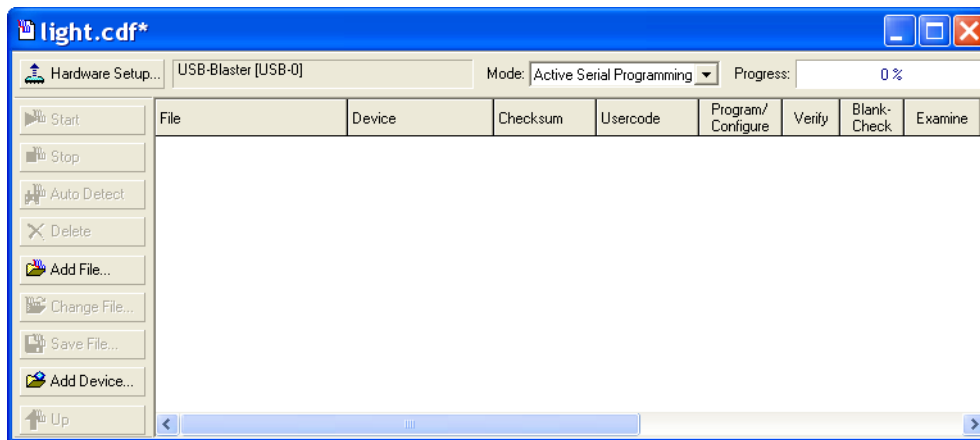


Figure 48. The Programmer window with Active Serial Programming selected.

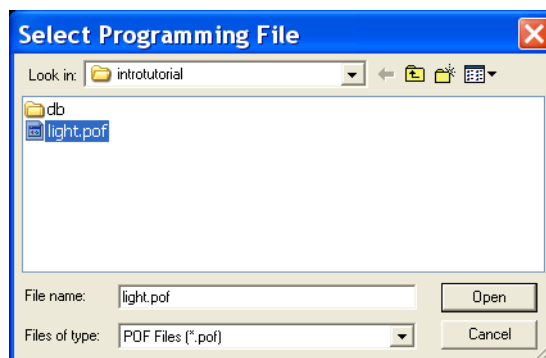


Figure 49. Choose the configuration file.

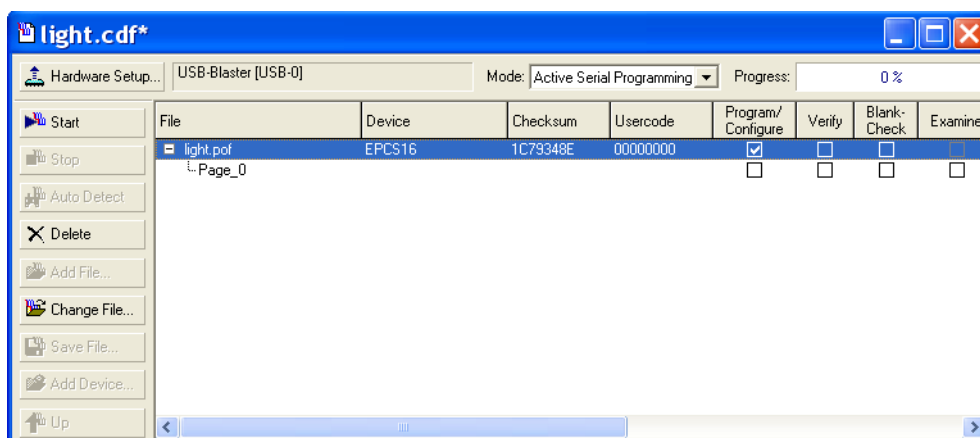


Figure 50. The updated Programmer window.

Flip the RUN/PROG switch on the DE2 board to the PROG position. Press Start in the window in Figure 50. An LED on the board will light up when the configuration data has been downloaded successfully. Also, the

Progress box in Figure 50 will indicate when the configuration and programming process is completed, as shown in Figure 51.

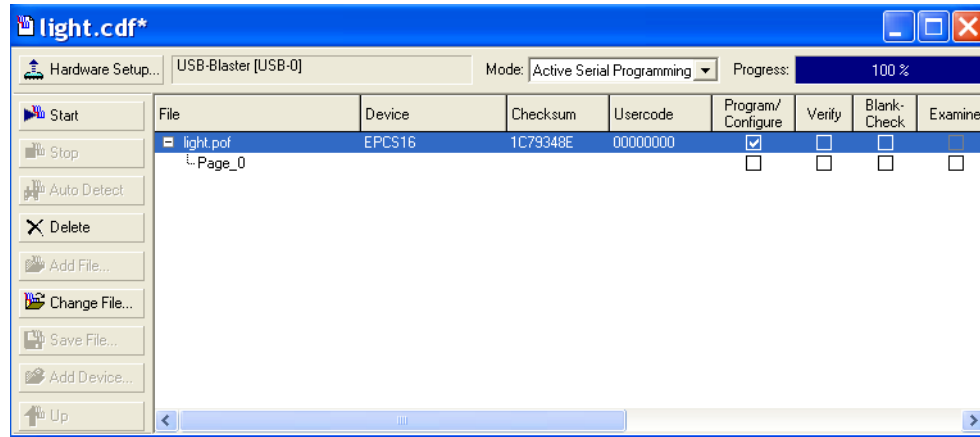


Figure 51. The Programmer window upon completion of programming.

8 Testing the Designed Circuit

Having downloaded the configuration data into the FPGA device, you can now test the implemented circuit. Flip the RUN/PROG switch to RUN position. Try all four valuations of the input variables x_1 and x_2 , by setting the corresponding states of the switches SW_1 and SW_0 . Verify that the circuit implements the truth table in Figure 12.

If you want to make changes in the designed circuit, first close the Programmer window. Then make the desired changes in the Verilog design file, compile the circuit, and program the board as explained above.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Quartus II Introduction Using Schematic Design

This tutorial presents an introduction to the Quartus[®] II CAD system. It gives a general overview of a typical CAD flow for designing circuits that are implemented by using FPGA devices, and shows how this flow is realized in the Quartus II software. The design process is illustrated by giving step-by-step instructions for using the Quartus II software to implement a very simple circuit in an Altera FPGA device.

The Quartus II system includes full support for all of the popular methods of entering a description of the desired circuit into a CAD system. This tutorial makes use of the schematic design entry method, in which the user draws a graphical diagram of the circuit. Two other versions of this tutorial are also available, which use the Verilog and VHDL hardware description languages, respectively.

The last step in the design process involves configuring the designed circuit in an actual FPGA device. To show how this is done, it is assumed that the user has access to the Altera DE2 Development and Education board connected to a computer that has Quartus II software installed. A reader who does not have access to the DE2 board will still find the tutorial useful to learn how the FPGA programming and configuration task is performed.

The screen captures in the tutorial were obtained using the Quartus II version 5.0; if other versions of the software are used, some of the images may be slightly different.

Contents:

- Typical CAD flow
- Getting started
- Starting a New Project
- Schematic Design Entry
- Compiling the Design
- Pin Assignment
- Simulating the Designed Circuit
- Programming and Configuring the FPGA Device
- Testing the Designed Circuit

Computer Aided Design (CAD) software makes it easy to implement a desired logic circuit by using a programmable logic device, such as a field-programmable gate array (FPGA) chip. A typical FPGA CAD flow is illustrated in Figure 1.

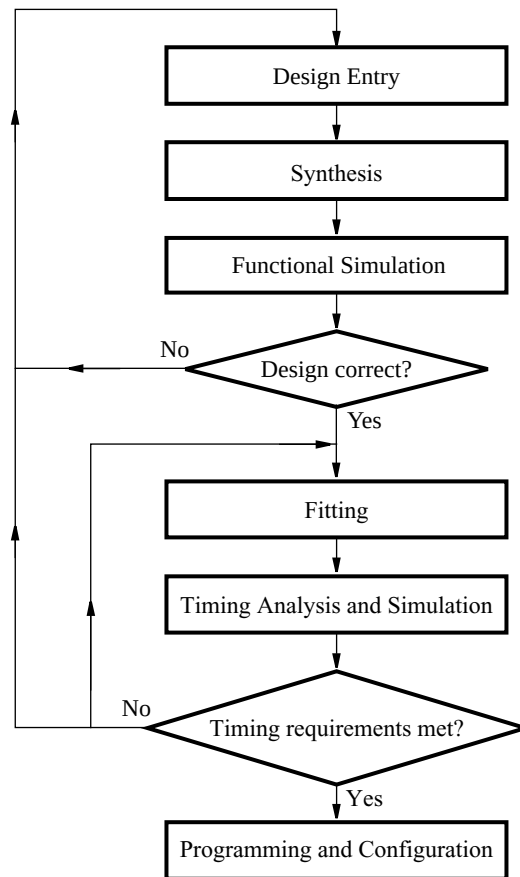


Figure 1. Typical CAD flow.

The CAD flow involves the following steps:

- **Design Entry** – the desired circuit is specified either by means of a schematic diagram, or by using a hardware description language, such as Verilog or VHDL.
- **Synthesis** – the entered design is synthesized into a circuit that consists of the logic elements (LEs) provided in the FPGA chip
- **Functional Simulation** – the synthesized circuit is tested to verify its functional correctness; this simulation does not take into account any timing issues

- **Fitting** – the CAD Fitter tool determines the placement of the LEs defined in the netlist into the LEs in an actual FPGA chip; it also chooses routing wires in the chip to make the required connections between specific LEs
- **Timing Analysis** – propagation delays along the various paths in the fitted circuit are analyzed to provide an indication of the expected performance of the circuit
- **Timing Simulation** – the fitted circuit is tested to verify both its functional correctness and timing
- **Programming and Configuration** – the designed circuit is implemented in a physical FPGA chip by programming the configuration switches that configure the LEs and establish the required wiring connections

This tutorial introduces the basic features of the Quartus II software. It shows how the software can be used to design and implement a circuit specified by means of a schematic diagram. It makes use of the graphical user interface to invoke the Quartus II commands. Doing this tutorial, the reader will learn about:

- Creating a project
- Entering a schematic diagram
- Synthesizing a circuit from the schematic diagram
- Fitting a synthesized circuit into an Altera FPGA
- Assigning the circuit inputs and outputs to specific pins on the FPGA
- Simulating the designed circuit
- Programming and configuring the FPGA chip on Altera's DE2 board

1 Getting Started

Each logic circuit, or subcircuit, being designed with Quartus II software is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory (folder) in the file system. To begin a new logic circuit design, the first step is to create a directory to hold its files. To hold the design files for this tutorial, we will use a directory *introtutorial*. The running example for this tutorial is a simple circuit for two-way light control.

Start the Quartus II software. You should see a display similar to the one in Figure 2. This display consists of several windows that provide access to all the features of Quartus II software, which the user selects with the computer mouse. Most of the commands provided by Quartus II software can be accessed by using a set of menus that are located below the title bar. For example, in Figure 2 clicking the left mouse button on the menu named File opens the menu shown in Figure 3. Clicking the left mouse button on the entry Exit exits from Quartus II software. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to press. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly.

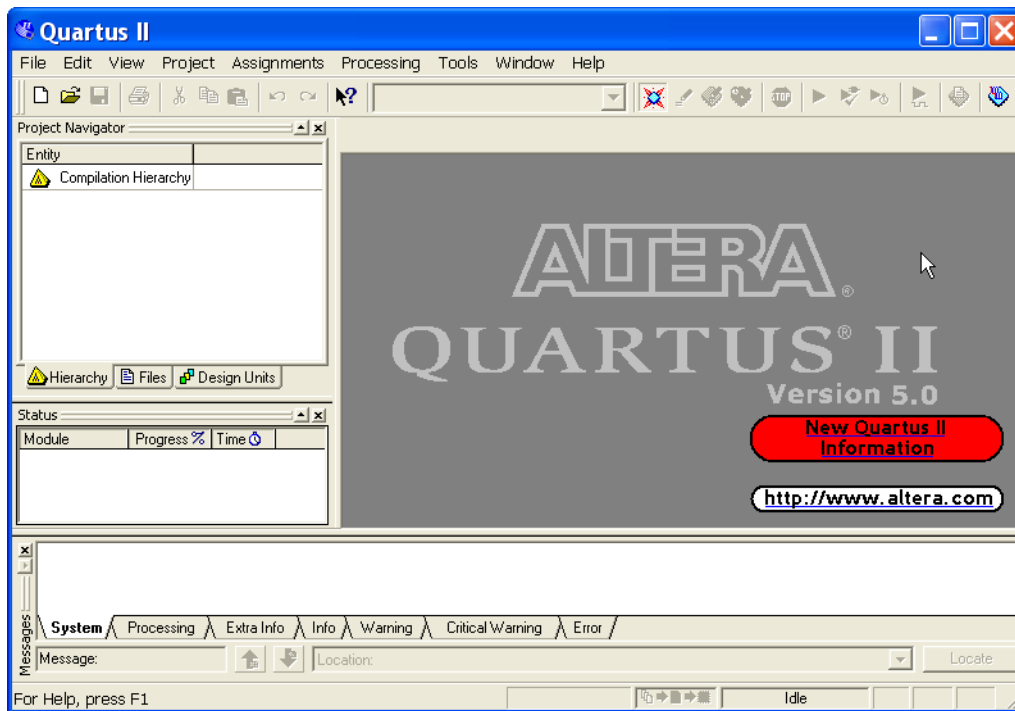


Figure 2. The main Quartus II display.

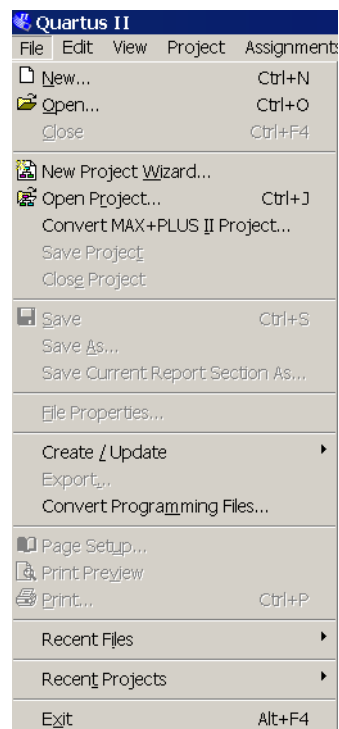


Figure 3. An example of the File menu.

For some commands it is necessary to access two or more menus in sequence. We use the convention **Menu1 > Menu2 > Item** to indicate that to select the desired command the user should first click the left mouse button on **Menu1**, then within this menu click on **Menu2**, and then within **Menu2** click on **Item**. For example, **File > Exit** uses the mouse to exit from the system. Many commands can be invoked by clicking on an icon displayed in one of the toolbars. To see the command associated with an icon, position the mouse over the icon and a tooltip will appear that displays the command name.

1.1 Quartus II Online Help

Quartus II software provides comprehensive online documentation that answers many of the questions that may arise when using the software. The documentation is accessed from the menu in the **Help** window. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the **Help** menu. For instance, selecting **Help > How to Use Help** gives an indication of what type of help is provided.

The user can quickly search through the Help topics by selecting **Help > Search**, which opens a dialog box into which key words can be entered. Another method, context-sensitive help, is provided for quickly finding documentation for specific topics. While using most applications, pressing the **F1** function key on the keyboard opens a Help display that shows the commands available for the application.

2 Starting a New Project

To start working on a new design we first have to define a new *design project*. Quartus II software makes the designer's task easy by providing support in the form of a *wizard*. Create a new project as follows:

1. Select **File > New Project Wizard** to reach the window in Figure 4, which indicates the capability of this wizard. You can skip this window in subsequent projects by checking the box **Don't show me this introduction again**. Press **Next** to get the window shown in Figure 5.

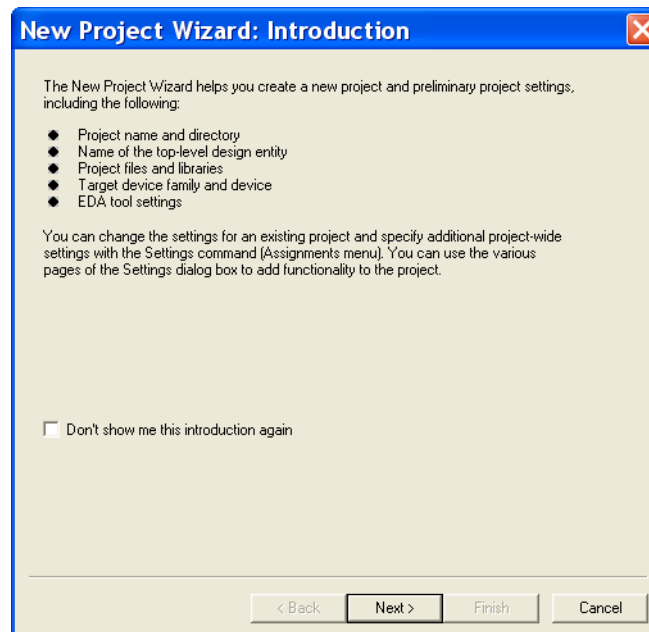


Figure 4. Tasks performed by the wizard.

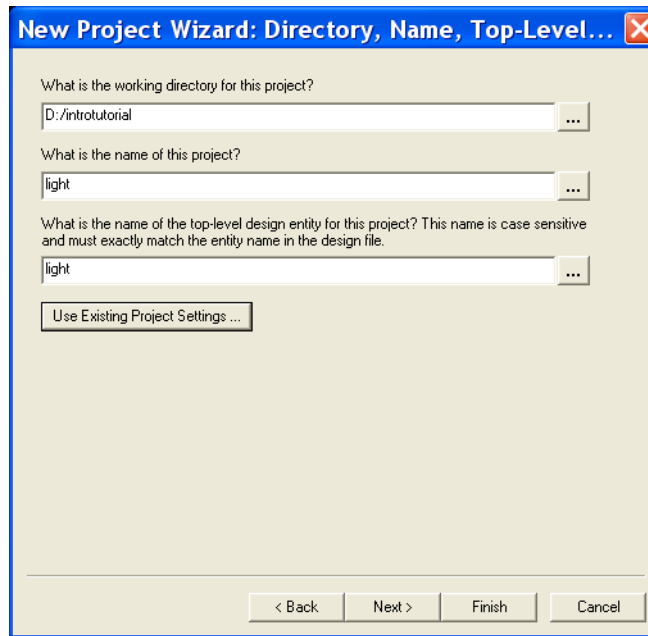


Figure 5. Creation of a new project.

2. Set the working directory to be *introtutorial*; of course, you can use some other directory name of your choice if you prefer. The project must have a name, which is usually the same as the top-level design entity that will be included in the project. Choose *light* as the name for both the project and the top-level entity, as shown in Figure 5. Press **Next**. Since we have not yet created the directory *introtutorial*, Quartus II software displays the pop-up box in Figure 6 asking if it should create the desired directory. Click **Yes**, which leads to the window in Figure 7.

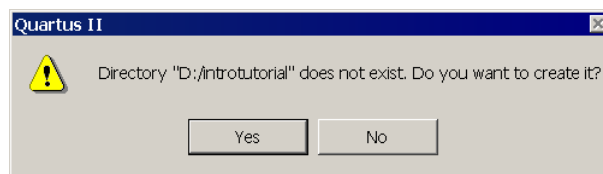


Figure 6. Quartus II software can create a new directory for the project.

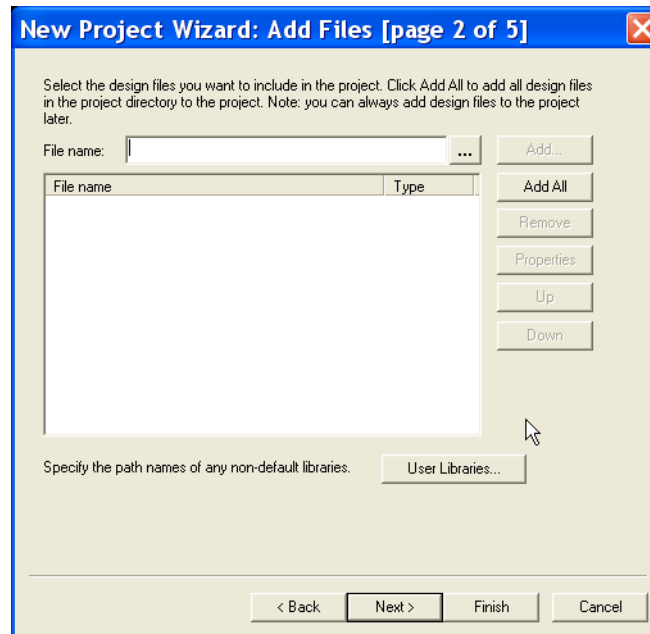


Figure 7. The wizard can include user-specified design files.

3. The wizard makes it easy to specify which existing files (if any) should be included in the project. Assuming that we do not have any existing files, click Next, which leads to the window in Figure 8.

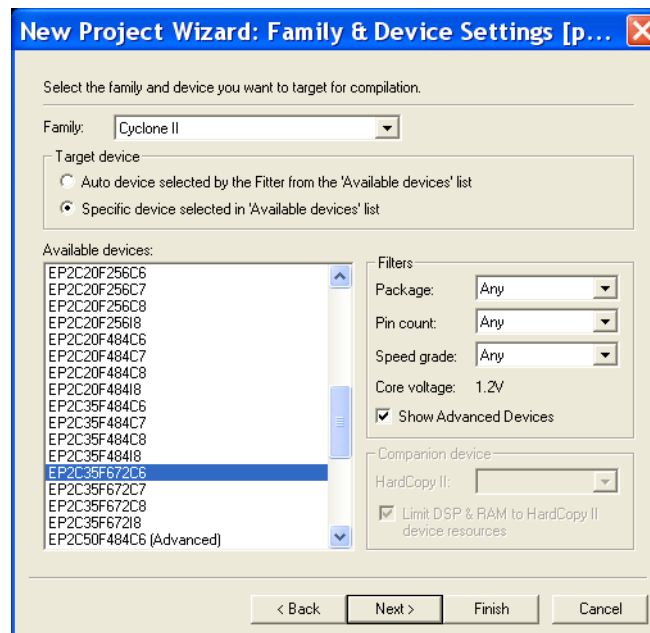


Figure 8. Choose the device family and a specific device.

4. We have to specify the type of device in which the designed circuit will be implemented. Choose Cyclone™ II as the target device family. We can let Quartus II software select a specific device in the family, or we can choose the device explicitly. We will take the latter approach. From the list of available devices, choose the device called EP2C35F672C6 which is the FPGA used on Altera's DE2 board. Press Next, which opens the window in Figure 9.

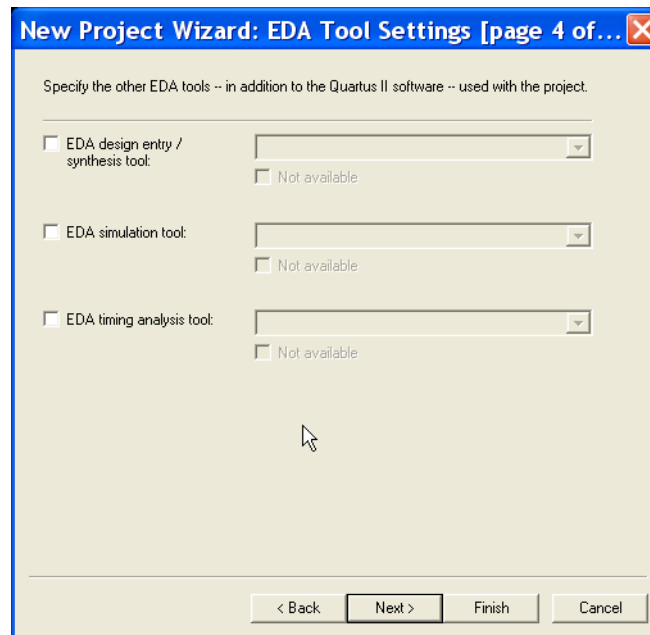


Figure 9. Other EDA tools can be specified.

5. The user can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is *EDA tools*, where the acronym stands for Electronic Design Automation. This term is used in Quartus II messages that refer to third-party tools, which are the tools developed and marketed by companies other than Altera. Since we will rely solely on Quartus II tools, we will not choose any other tools. Press Next.
6. A summary of the chosen settings appears in the screen shown in Figure 10. Press Finish, which returns to the main Quartus II window, but with *light* specified as the new project, in the display title bar, as indicated in Figure 11.

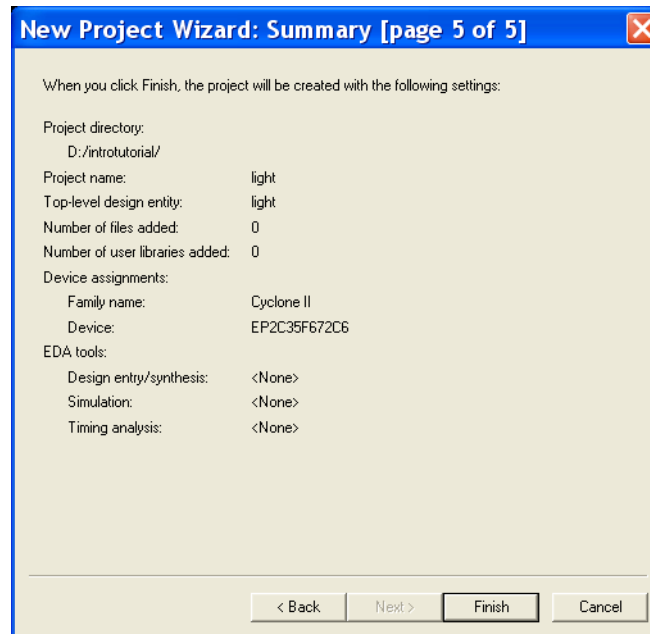


Figure 10. Summary of the project settings.

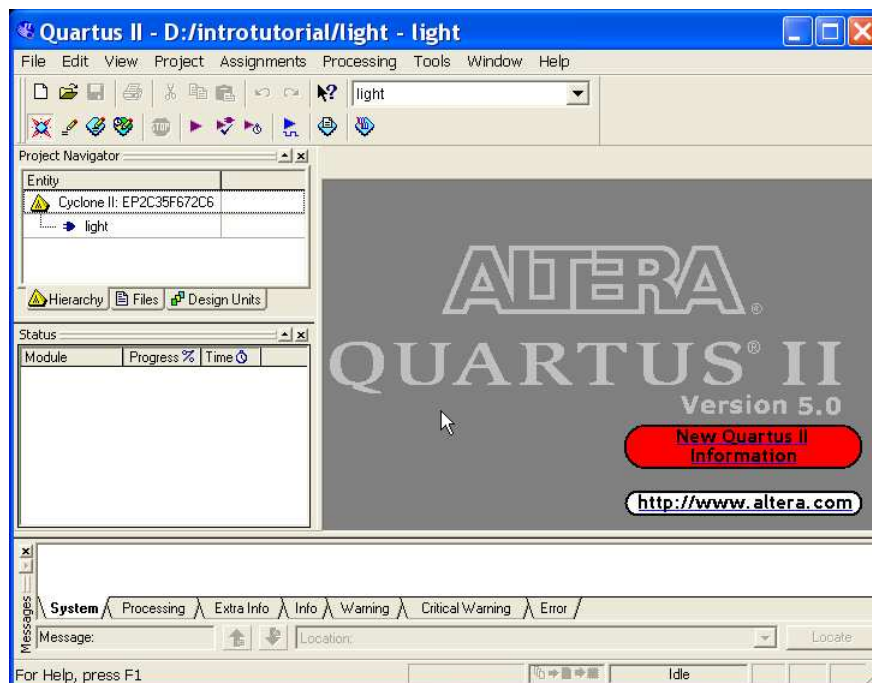


Figure 11. The Quartus II display for the created project.

3 Design Entry Using the Graphic Editor

As a design example, we will use the two-way light controller circuit shown in Figure 12. The circuit can be used to control a single light from either of the two switches, x_1 and x_2 , where a closed switch corresponds to the logic value 1. The truth table for the circuit is also given in the figure. Note that this is just the Exclusive-OR function of the inputs x_1 and x_2 , but we will implement it using the gates shown.

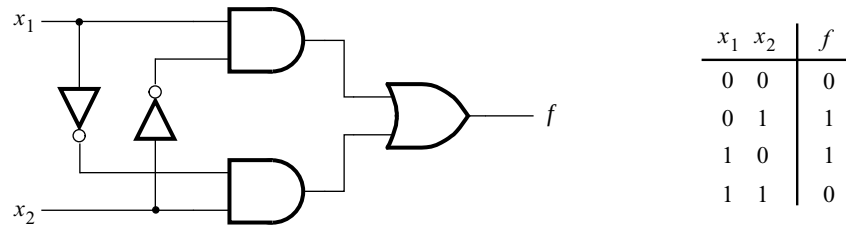


Figure 12. The light controller circuit.

The Quartus II Graphic Editor can be used to specify a circuit in the form of a block diagram. Select File > New to get the window in Figure 13, choose Block Diagram/Schematic File, and click OK. This opens the Graphic Editor window. The first step is to specify a name for the file that will be created. Select File > Save As to open the pop-up box depicted in Figure 14. In the box labeled Save as type choose Block Diagram/Schematic File (*.bdf). In the box labeled File name type *light*, to match the name given in Figure 5, which was specified when the project was created. Put a checkmark in the box Add file to current project. Click Save, which puts the file into the directory *introtutorial* and leads to the Graphic Editor window displayed in Figure 15.

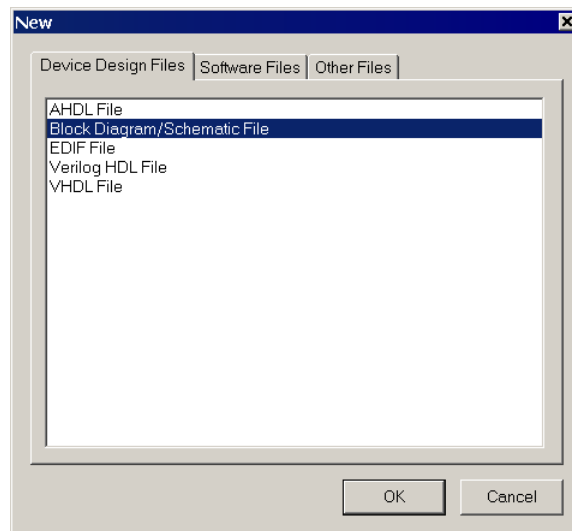


Figure 13. Choose to prepare a block diagram.

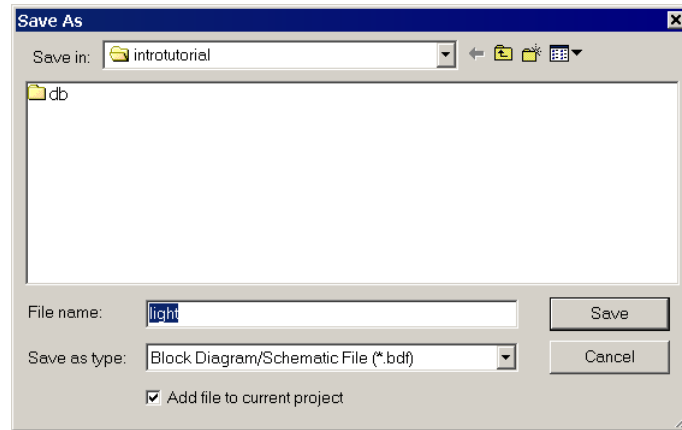


Figure 14. Name the file.



Figure 15. Graphic Editor window.

3.1 Importing Logic-Gate Symbols

The Graphic Editor provides a number of libraries which include circuit elements that can be imported into a schematic. Double-click on the blank space in the Graphic Editor window, or click on the  icon in the toolbar that looks like an AND gate. A pop-up box in Figure 16 will appear. Expand the hierarchy in the Libraries box as shown in the figure. First expand *libraries*, then expand the library *primitives*, followed by expanding the library *logic* which comprises the logic gates. Select *and2*, which is a two-input AND gate, and click OK. Now, the AND gate symbol will appear in the Graphic Editor window. Using the mouse, move the symbol to a desirable location and click to place it there. Import the second AND gate, which can be done simply by positioning the mouse pointer over the existing AND-gate symbol, right-clicking, and dragging to make a copy of the symbol. A symbol in the Graphic Editor window can be moved by clicking on it and dragging it to a new location with the mouse

button pressed. Next, select *or2* from the library and import the OR gate into the diagram. Then, select *not* and import two instances of the NOT gate. Rotate the NOT gates into proper position by using the “Rotate left 90” icon . Arrange the gates as shown in Figure 17.

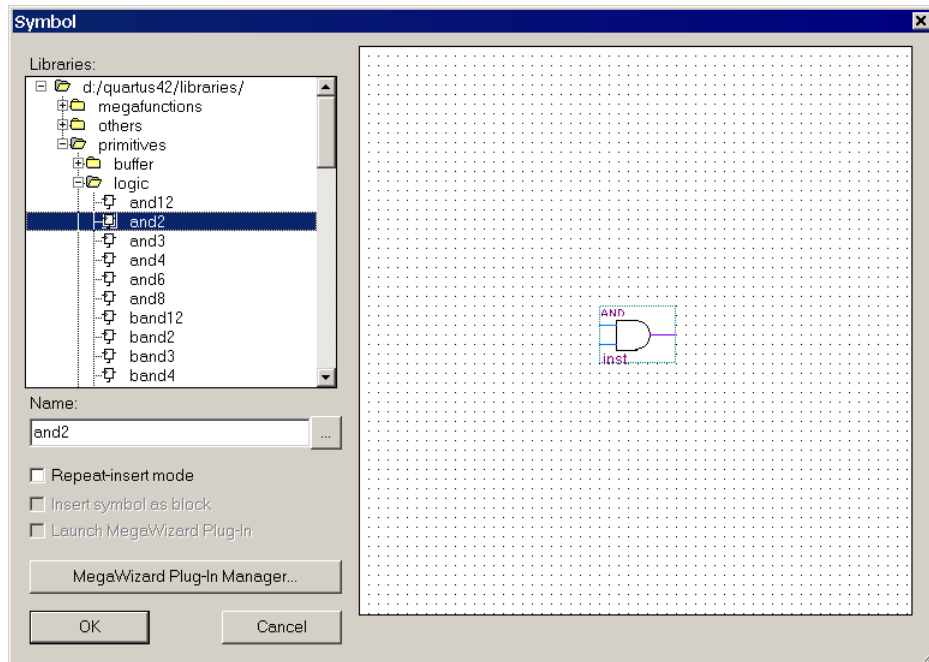


Figure 16. Choose a symbol from the library.

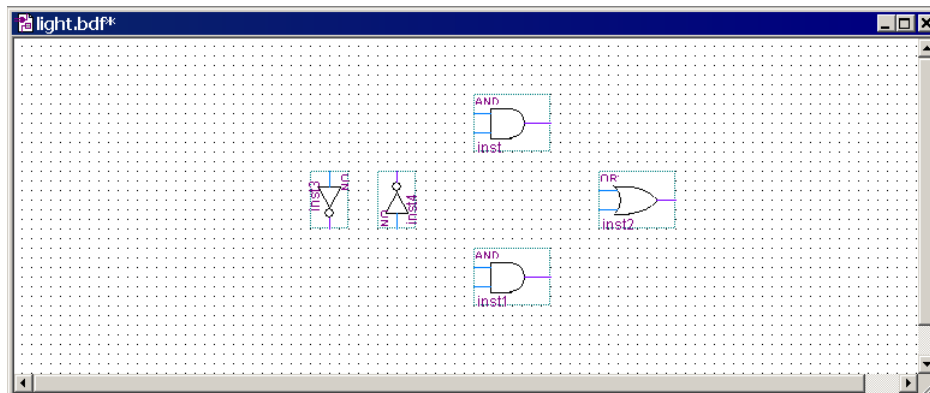


Figure 17. Import the gate symbols into the Graphic Editor window.

3.2 Importing Input and Output Symbols

Having entered the logic-gate symbols, it is now necessary to enter the symbols that represent the input and output ports of the circuit. Use the same procedure as for importing the gates, but choose the port symbols from the library *primitives/pin*. Import two instances of the input port and one instance of the output port, to obtain the image in Figure 18.

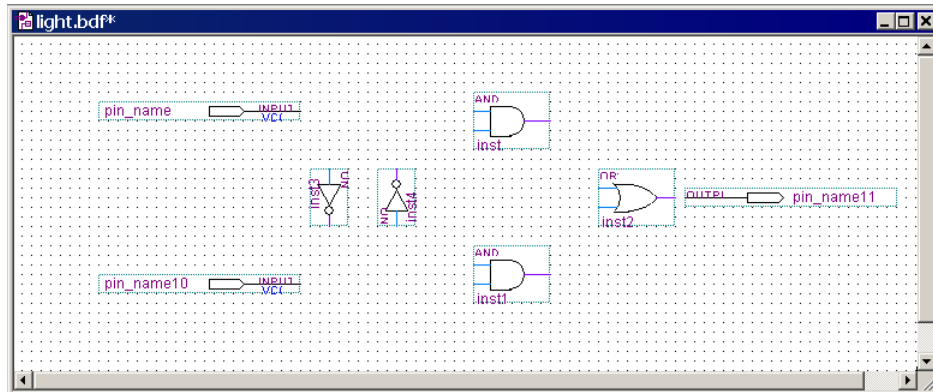


Figure 18. Import the input and output pins.

Assign names to the input and output symbols as follows. Point to the word *pin_name* on the top input symbol and double-click the mouse. The dialog box in Figure 19 will appear. Type the pin name, *x1*, and click OK. Similarly, assign the name *x2* to the other input and *f* to the output.

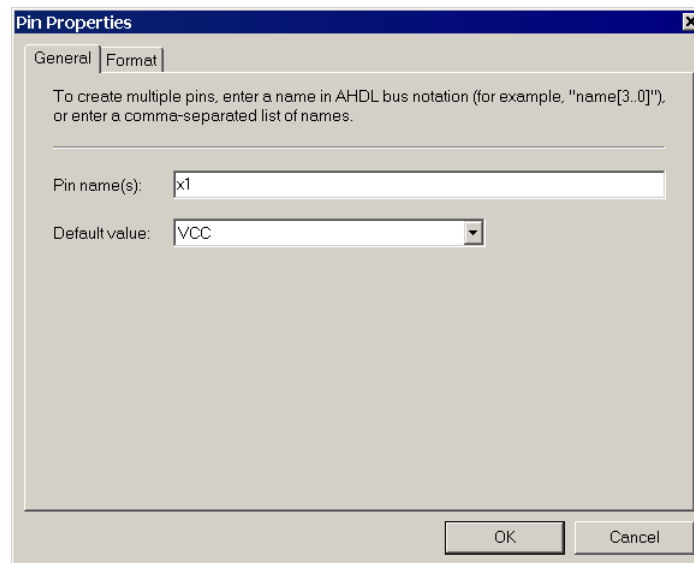


Figure 19. Naming of a pin.

3.3 Connecting Nodes with Wires

The symbols in the diagram have to be connected by drawing lines (wires). Click on the icon  in the toolbar to activate the Orthogonal Node Tool. Position the mouse pointer over the right edge of the *x1* input pin. Click and hold the mouse button and drag the mouse to the right until the drawn line reaches the pinstub on the top input of the AND gate. Release the mouse button, which leaves the line connecting the two pinstubs. Next, draw a wire from the input pinstub of the leftmost NOT gate to touch the wire that was drawn above it. Note that a dot will appear indicating a connection between the two wires.

Use the same procedure to draw the remaining wires in the circuit. If a mistake is made, a wire can be selected by clicking on it, and removed by pressing the Delete key on the keyboard. Upon completing the diagram, click

on the icon , to activate the Selection and Smart Drawing Tool. Now, changes in the appearance of the diagram can be made by selecting a particular symbol or wire and either moving it to a different location or deleting it. The final diagram is shown in Figure 20; save it.

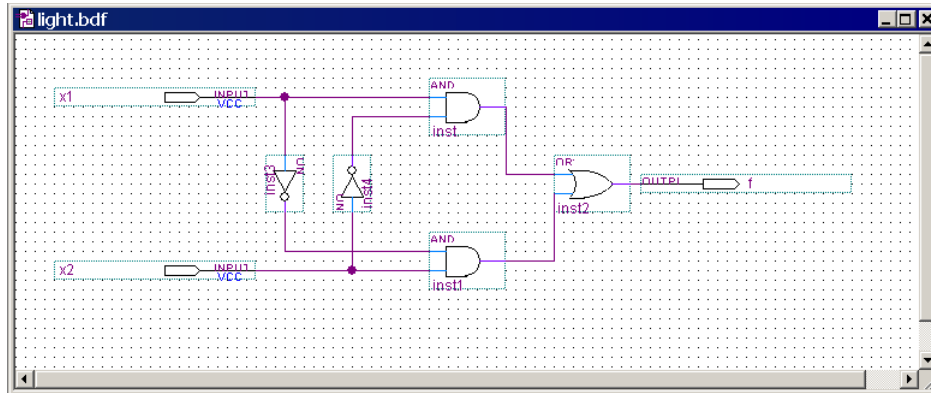


Figure 20. The completed schematic diagram.

4 Compiling the Designed Circuit

The entered schematic diagram file, *light.bdf*, is processed by several Quartus II tools that analyze the file, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the *Compiler*.

Run the Compiler by selecting **Processing > Start Compilation**, or by clicking on the toolbar icon  that looks like a purple triangle. As the compilation moves through various stages, its progress is reported in a window on the left side of the Quartus II display. Successful (or unsuccessful) compilation is indicated in a pop-up box. Acknowledge it by clicking OK, which leads to the Quartus II display in Figure 21. In the message window, at the bottom of the figure, various messages are displayed. In case of errors, there will be appropriate messages given.

When the compilation is finished, a compilation report is produced. A window showing this report is opened automatically, as seen in Figure 21. The window can be resized, maximized, or closed in the normal way, and it can be opened at any time either by selecting **Processing > Compilation Report** or by clicking on the icon . The report includes a number of sections listed on the left side of its window. Figure 21 displays the Compiler Flow Summary section, which indicates that only one logic element and three pins are needed to implement this tiny circuit on the selected FPGA chip. Another section is shown in Figure 22. It is reached by selecting **Analysis & Synthesis > Equations** on the left side of the compilation report. Here we see the logic expressions produced by the Compiler when synthesizing the designed circuit. Observe that *f* is the output derived as

$$f = x1 \$ x2$$

where the \$ sign is used to represent the Exclusive-OR operation. Obviously, the Compiler recognized that the functionality of the circuit in our design file, *light.bdf*, can be represented by this expression.

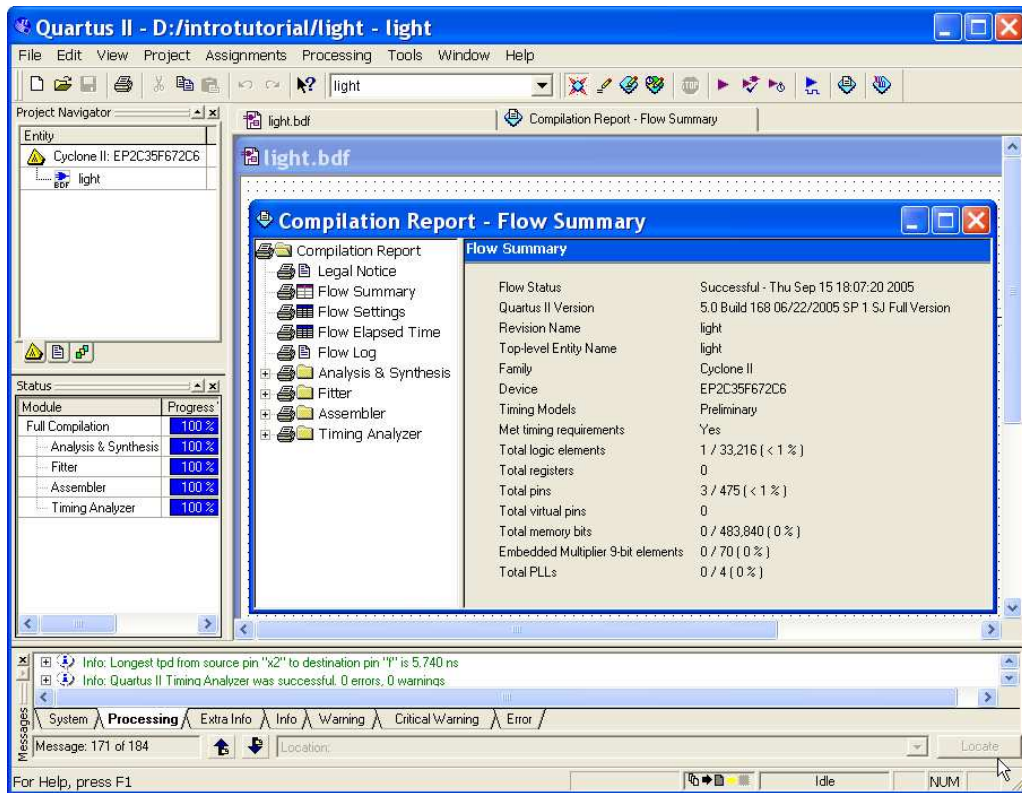


Figure 21. Display after a successful compilation.

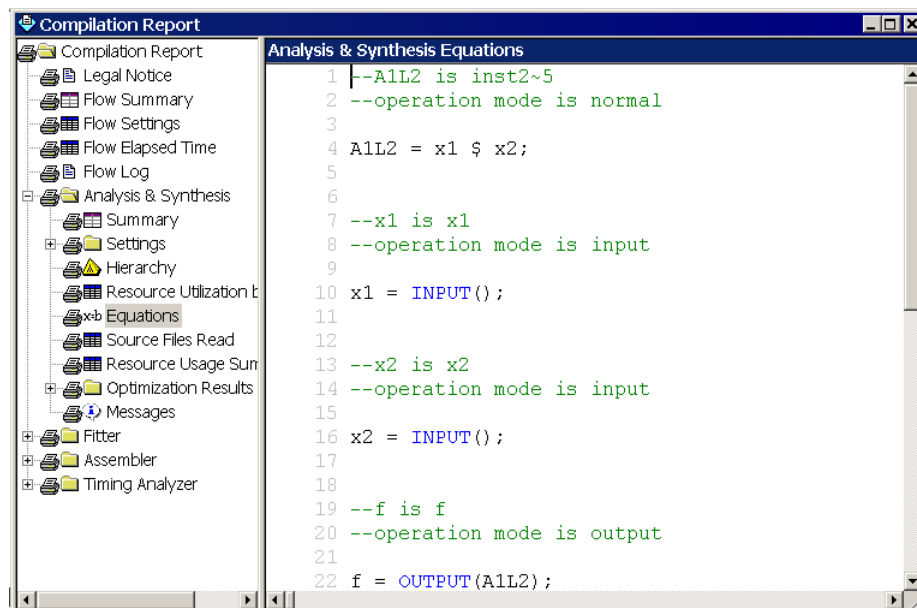


Figure 22. Compilation report showing the synthesized equations.

4.1 Errors

Quartus II software displays messages produced during compilation in the Messages window. If the block diagram design file is correct, one of the messages will state that the compilation was successful and that there are no errors.

If the Compiler does not report zero errors, then there is at least one mistake in the schematic entry. In this case a message corresponding to each error found will be displayed in the Messages window. Double-clicking on an error message will highlight the offending part of the circuit in the Graphic Editor window. Similarly, the Compiler may display some warning messages. Their details can be explored in the same way as in the case of error messages. The user can obtain more information about a specific error or warning message by selecting the message and pressing the F1 function key.

To see the effect of an error, open the file *light.bdf*. Remove the wire connecting the output of the top AND gate to the OR gate. To do this, click on the  icon, click the mouse on the wire to be removed (to select it) and press Delete. Compile the erroneous design by clicking on the  icon. A pop-up box will ask if the changes made to the *light.bdf* file should be saved; click Yes. After trying to compile the circuit, Quartus II software will display a pop-up box indicating that the compilation was not successful. Acknowledge it by clicking OK. The compilation report summary, given in Figure 23, now confirms the failed result. Expand the Analysis & Synthesis part of the report and then select Messages to have the messages displayed as shown in Figure 24. Double-click on the first error message, which states that one of the nodes is missing a source. Quartus II software responds by displaying the *light.bdf* schematic and highlighting the OR gate which is affected by the error, as shown in Figure 25. Correct the error and recompile the design.

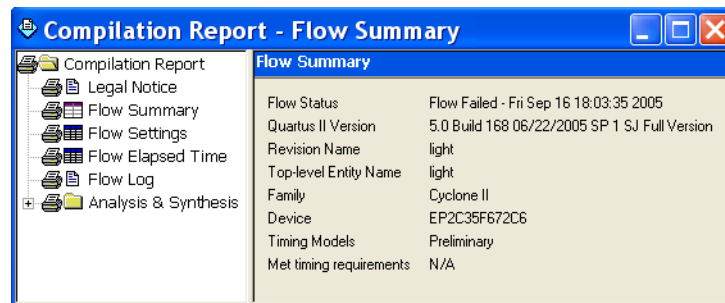


Figure 23. Compilation report for the failed design.

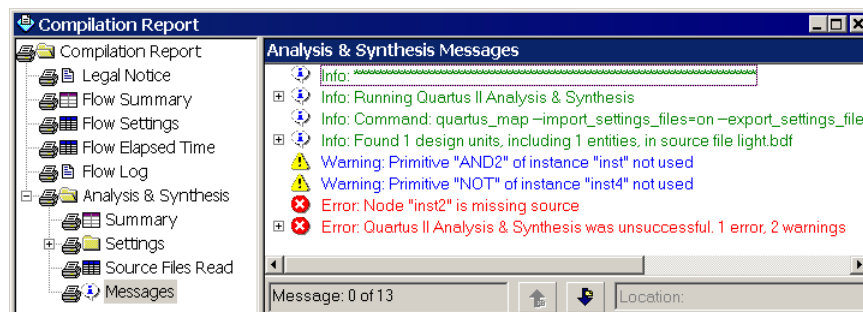


Figure 24. Error messages.

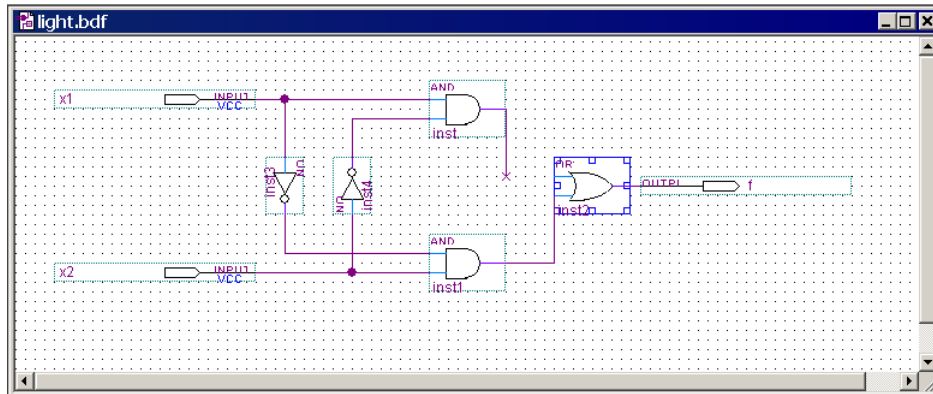


Figure 25. Identifying the location of the error.

5 Pin Assignment

During the compilation above, the Quartus II Compiler was free to choose any pins on the selected FPGA to serve as inputs and outputs. However, the DE2 board has hardwired connections between the FPGA pins and the other components on the board. We will use two toggle switches, labeled SW_0 and SW_1 , to provide the external inputs, x_1 and x_2 , to our example circuit. These switches are connected to the FPGA pins N25 and N26, respectively. We will connect the output f to the green light-emitting diode labeled $LEDG_0$, which is hardwired to the FPGA pin AE22.

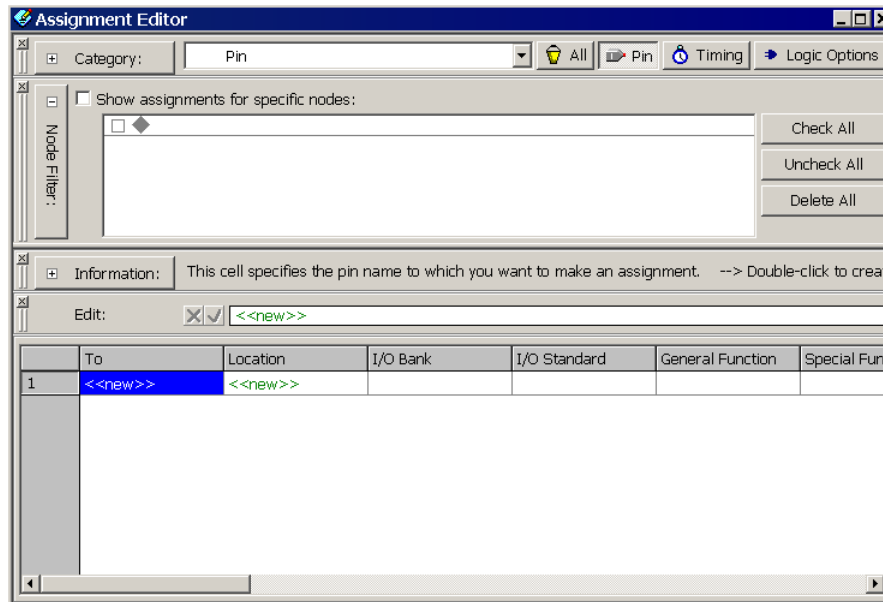


Figure 26. The Assignment Editor window.

Pin assignments are made by using the Assignment Editor. Select **Assignments > Pins** to reach the window in Figure 26. Under **Category** select **Pin**. Double-click on the entry **<<new>>** which is highlighted in blue in the column labeled **To**. The drop-down menu in Figure 27 will appear. Click on x_1 as the first pin to be assigned; this will enter x_1 in the displayed table. Follow this by double-clicking on the box to the right of this new x_1

entry, in the column labeled Location. Now, the drop-down menu in Figure 28 appears. Scroll down and select PIN_N25. Instead of scrolling down the menu to find the desired pin, you can just type the name of the pin (N25) in the Location box. Use the same procedure to assign input x_2 to pin N26 and output f to pin AE22, which results in the image in Figure 29. To save the assignments made, choose File > Save. You can also simply close the Assignment Editor window, in which case a pop-up box will ask if you want to save the changes to assignments; click Yes. Recompile the circuit, so that it will be compiled with the correct pin assignments.

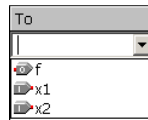


Figure 27. The drop-down menu displays the input and output names.

Location	I/O Bank	I/O Standard	General Function	Special Fun
		LVTTTL		
PIN_N1	I/O Bank 2	Dedicated Clock	CLK1, LVDSCLK0n, Input	
PIN_N2	I/O Bank 2	Dedicated Clock	CLK0, LVDSCLK0p, Input	
PIN_N9	I/O Bank 2	Row I/O	LVDS31p	
PIN_N18	I/O Bank 5	Row I/O	LVDS110p	
PIN_N20	I/O Bank 5	Row I/O	LVDS124p	
PIN_N23	I/O Bank 5	Row I/O	LVDS126p, DPCLK7/DQS0R/CQ1R	
PIN_N24	I/O Bank 5	Row I/O	LVDS126n	
PIN_N25	I/O Bank 5	Dedicated Clock	CLK4, LVDSCLK2p, Input	
PIN_N26	I/O Bank 5	Dedicated Clock	CLK5, LVDSCLK2n, Input	
PIN_P1	I/O Bank 1	Dedicated Clock	CLK3, LVDSCLK1n, Input	
PIN_P2	I/O Bank 1	Dedicated Clock	CLK2, LVDSCLK1p, Input	
PIN_P3	I/O Bank 1	Row I/O	LVDS26p, DPCLK1/DQS1L/CQ1L#	
PIN_P4	I/O Bank 1	Row I/O	LVDS26n	
PIN_P6	I/O Bank 1	Row I/O	LVDS22n	
PIN_P7	I/O Bank 1	Row I/O	LVDS22p	
PIN_P9	I/O Bank 2	Row I/O	LVDS31n	
PIN_P17	I/O Bank 6	Row I/O	LVDS130n	
PIN_P18	I/O Bank 5	Row I/O	LVDS110n	
PIN_P23	I/O Bank 6	Row I/O	LVDS127p, DPCLK6/DQS1R/CQ1R#	
PIN_P24	I/O Bank 6	Row I/O	LVDS127n	

Figure 28. The available pins.

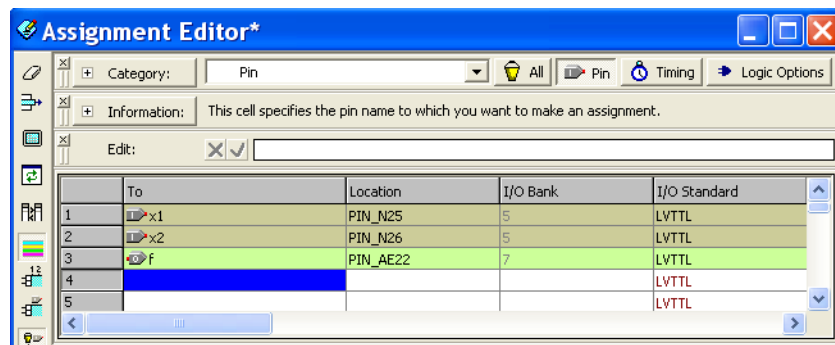


Figure 29. The complete assignment.

The DE2 board has fixed pin assignments. Having finished one design, the user will want to use the same pin assignment for subsequent designs. Going through the procedure described above becomes tedious if there are many pins used in the design. A useful Quartus II feature allows the user to both export and import the pin assignments from a special file format, rather than creating them manually using the Assignment Editor. A simple

file format that can be used for this purpose is the *comma separated value (CSV)* format, which is a common text file format that contains comma-delimited values. This file format is often used in conjunction with the Microsoft Excel spreadsheet program, but the file can also be created by hand using any plain ASCII text editor. The format for the file for our simple project is

To, Location
x1, PIN_N25
x2, PIN_N26
f, PIN_AE22

By adding lines to the file, any number of pin assignments can be created. Such csv files can be imported into any design project.

If you created a pin assignment for a particular project, you can export it for use in a different project. To see how this is done, open again the Assignment Editor to reach the window in Figure 29. Now, select **File > Export** which leads to the window in Figure 30. Here, the file *light.csv* is available for export. Click on **Export**. If you now look in the directory *introtutorial*, you will see that the file *light.csv* has been created.

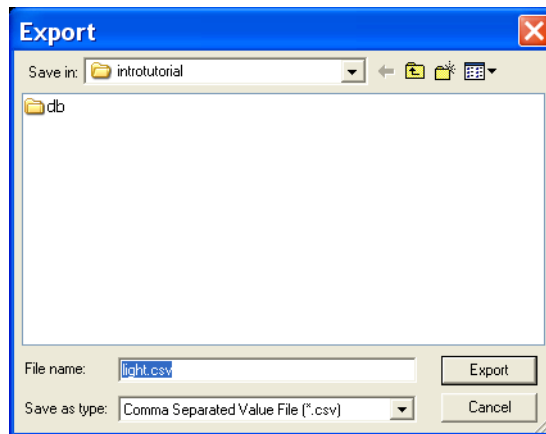


Figure 30. Exporting the pin assignment.

You can import a pin assignment by choosing **Assignments > Import Assignments**. This opens the dialogue in Figure 31 to select the file to import. Type the name of the file, including the csv extension and the full path to the directory that holds the file, in the File Name box and press **OK**. Of course, you can also browse to find the desired file.

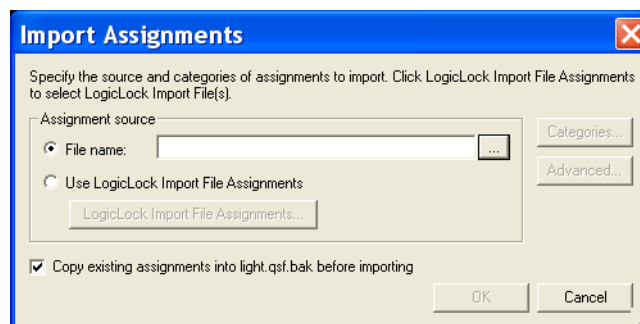


Figure 31. Importing the pin assignment.

For convenience when using large designs, all relevant pin assignments for the DE2 board are given in the file called *DE2_pin_assignments.csv* in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages. This file uses the names found in the *DE2 User Manual*. If we wanted to make the pin assignments for our example circuit by importing this file, then we would have to use the same names in our Block Diagram/Schematic design file; namely, *SW[0]*, *SW[1]* and *LEDG[0]* for *x1*, *x2* and *f*, respectively. Since these signals are specified in the *DE2_pin_assignments.csv* file as elements of vectors *SW* and *LEDG*, we must refer to them in the same way in our design file. For example, in the *DE2_pin_assignments.csv* file the 18 toggle switches are called *SW[17]* to *SW[0]*. In a design file they can also be referred to as a vector *SW[17..0]*.

6 Simulating the Designed Circuit

Before implementing the designed circuit in the FPGA chip on the DE2 board, it is prudent to simulate it to ascertain its correctness. Quartus II software includes a simulation tool that can be used to simulate the behavior of a designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. It is also necessary to specify which outputs, as well as possible internal points in the circuit, the designer wishes to observe. The simulator applies the test vectors to a model of the implemented circuit and determines the expected response. We will use the Quartus II Waveform Editor to draw the test vectors, as follows:

1. Open the Waveform Editor window by selecting **File > New**, which gives the window shown in Figure 32. Click on the **Other Files** tab to reach the window displayed in Figure 33. Choose **Vector Waveform File** and click OK.

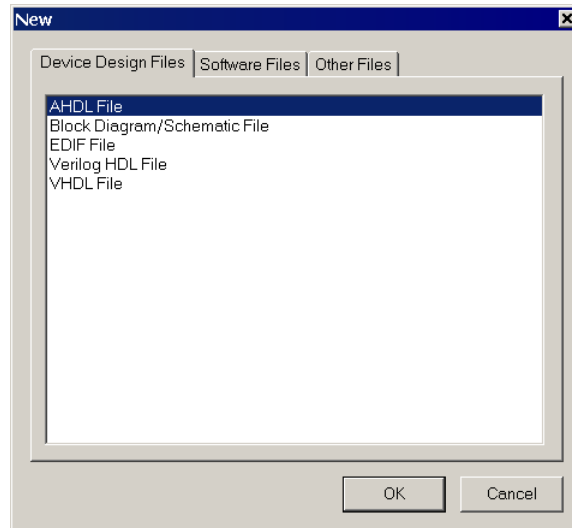


Figure 32. Need to prepare a new file.

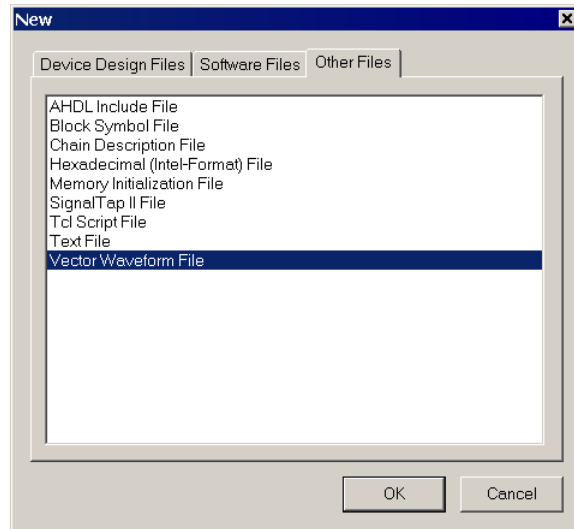


Figure 33. Choose to prepare a test-vector file.

2. The Waveform Editor window is depicted in Figure 34. Save the file under the name *light.vwf*; note that this changes the name in the displayed window. Set the desired simulation to run from 0 to 200 ns by selecting **Edit > End Time** and entering 200 ns in the dialog box that pops up. Selecting **View > Fit in Window** displays the entire simulation range of 0 to 200 ns in the window, as shown in Figure 35. You may wish to resize the window to its maximum size.

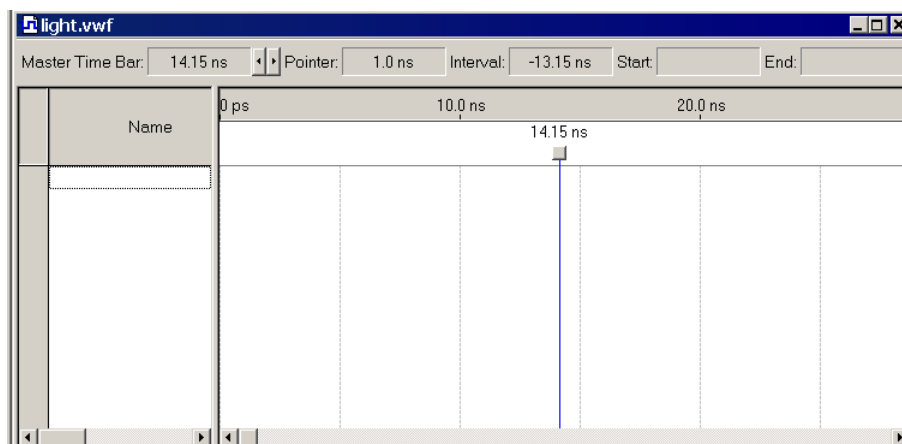


Figure 34. The Waveform Editor window.

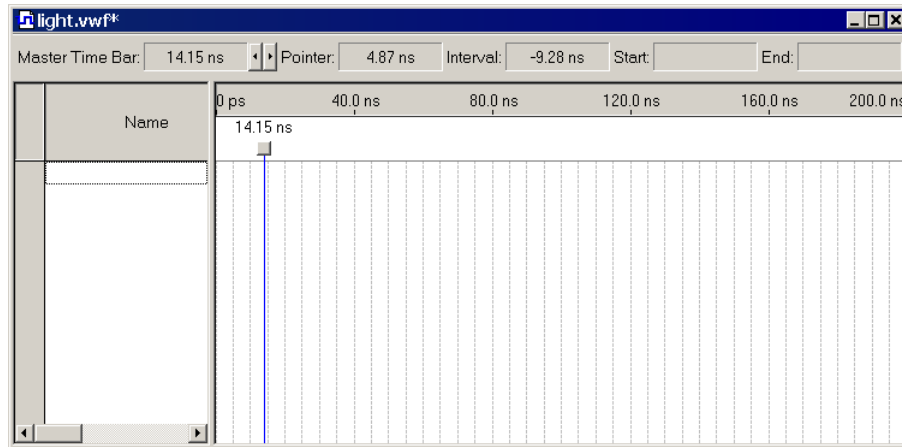


Figure 35. The augmented Waveform Editor window.

- Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert Node or Bus** to open the window in Figure 36. It is possible to type the name of a signal (pin) into the Name box, but it is easier to click on the button labeled **Node Finder** to open the window in Figure 37. The Node Finder utility has a filter used to indicate what type of nodes are to be found. Since we are interested in input and output pins, set the filter to **Pins: all**. Click the **List** button to find the input and output nodes as indicated on the left side of the figure.

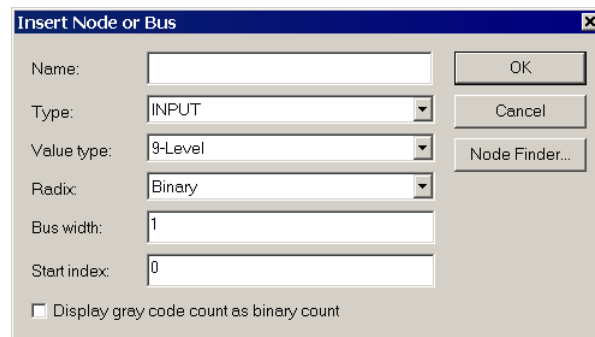


Figure 36. The Insert Node or Bus dialogue.

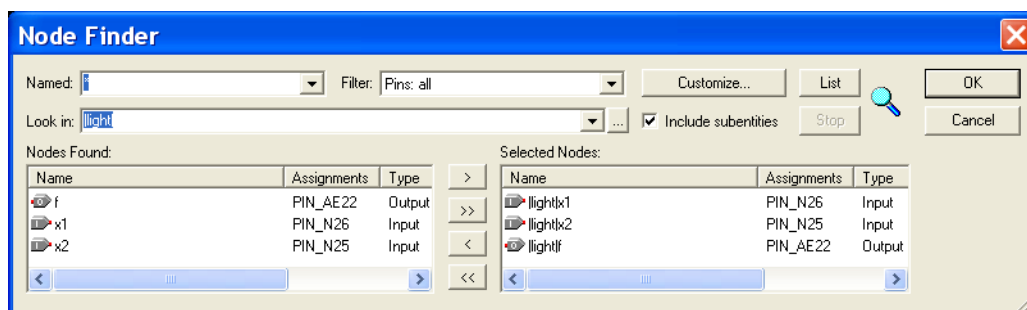


Figure 37. Selecting nodes to insert into the Waveform Editor.

Click on the $x1$ signal in the Nodes Found box in Figure 37, and then click the > sign to add it to the Selected Nodes box on the right side of the figure. Do the same for $x2$ and f . Click OK to close the Node Finder window, and then click OK in the window of Figure 36. This leaves a fully displayed Waveform Editor window, as shown in Figure 38. If you did not select the nodes in the same order as displayed in Figure 38, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click on the node name (in the Name column) and release the mouse button. The waveform is now highlighted to show the selection. Click again on the waveform and drag it up or down in the Waveform Editor.

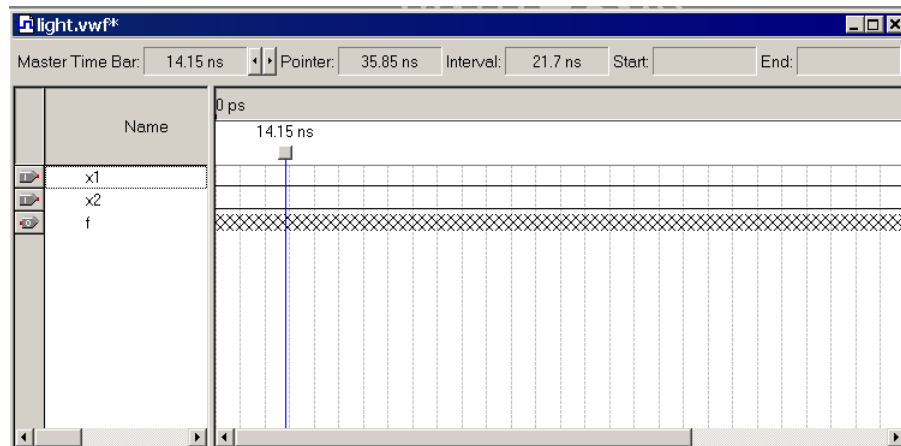


Figure 38. The nodes needed for simulation.

4. We will now specify the logic values to be used for the input signals $x1$ and $x2$ during simulation. The logic values at the output f will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing View > Snap to Grid). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This reference line is used in analyzing the timing of a circuit; move it to the $time = 0$ position. The waveforms can be drawn using the Selection Tool, which is activated by selecting the icon  in the toolbar, or the Waveform Editing Tool, which is activated by the icon .

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. In a large circuit the number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations. However, for our tiny circuit we can simulate all four input valuations given in Figure 12. We will use four 50-ns time intervals to apply the four test vectors.

We can generate the desired input waveforms as follows. Click on the waveform name for the $x1$ node. Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for setting a selected signal to 0, 1, unknown (X), high impedance (Z), don't care (DC), inverting its existing value (INV), or defining a clock waveform. Each command can be activated by using the Edit > Value command, or via the toolbar for the Waveform Editor. The Edit menu can also be opened by right-clicking on a waveform name.

Set $x1$ to 0 in the time interval 0 to 100 ns, which is probably already set by default. Next, set $x1$ to 1 in the time interval 100 to 200 ns. Do this by pressing the mouse at the start of the interval and dragging it to its end, which highlights the selected interval, and choosing the logic value 1 in the toolbar. Make $x2 = 1$ from 50 to 100 ns and also from 150 to 200 ns, which corresponds to the truth table in Figure 12. This should

produce the image in Figure 39. Observe that the output f is displayed as having an unknown value at this time, which is indicated by a hashed pattern; its value will be determined during simulation. Save the file.

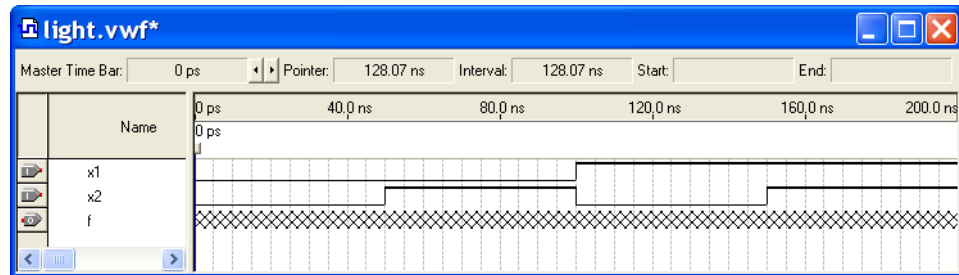


Figure 39. Setting of test values.

6.1 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and interconnection wires in the FPGA are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit.

6.1.1 Functional Simulation

To perform the functional simulation, select **Assignments > Settings** to open the Settings window. On the left side of this window click on **Simulator** to display the window in Figure 40, choose **Functional** as the simulation mode, and click OK. The Quartus II simulator takes the inputs and generates the outputs defined in the *light.vwf* file. Before running the functional simulation it is necessary to create the required netlist, which is done by selecting **Processing > Generate Functional Simulation Netlist**. A simulation run is started by **Processing > Start Simulation**, or by using the icon . At the end of the simulation, Quartus II software indicates its successful completion and displays a Simulation Report illustrated in Figure 41. If your report window does not show the entire simulation time range, click on the report window to select it and choose **View > Fit in Window**. Observe that the output f is as specified in the truth table of Figure 12.

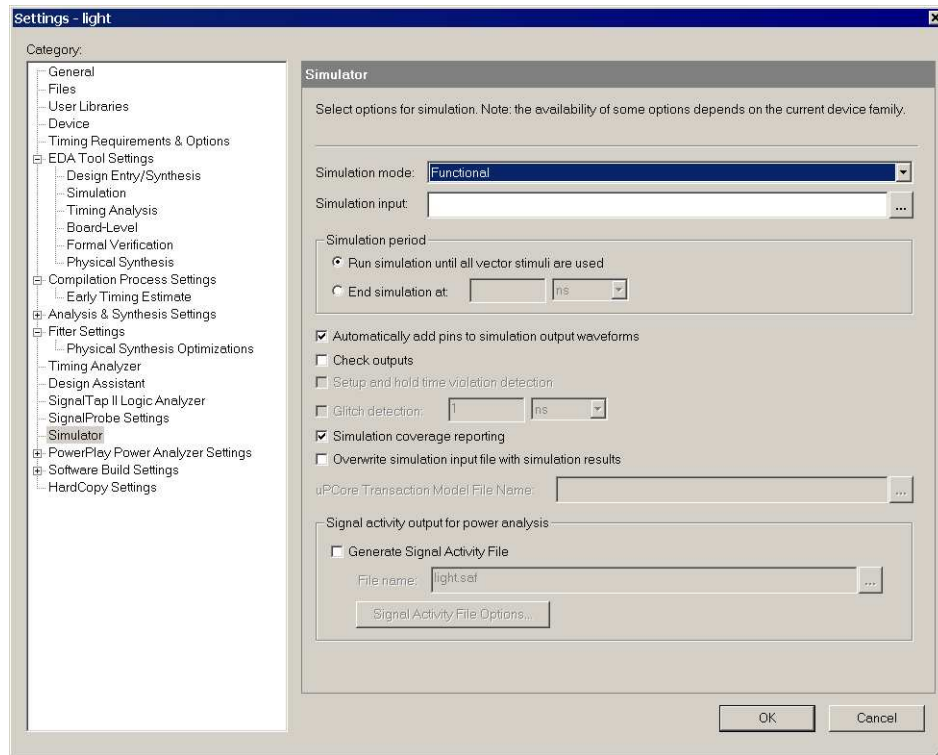


Figure 40. Specifying the simulation mode.

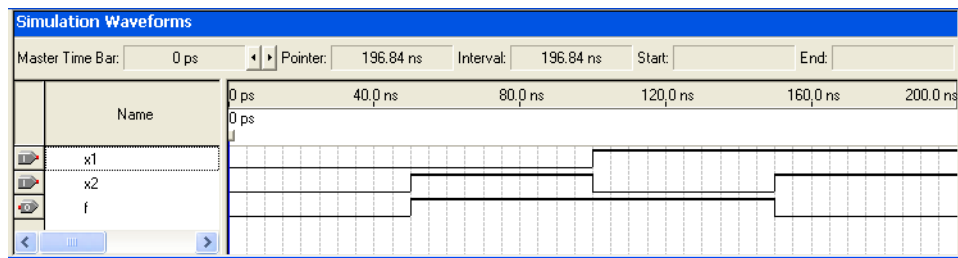


Figure 41. The result of functional simulation.

6.1.2 Timing Simulation

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how it will behave when it is actually implemented in the chosen FPGA device. Select **Assignments > Settings > Simulator** to get to the window in Figure 40, choose **Timing** as the simulation mode, and click **OK**. Run the simulator, which should produce the waveforms in Figure 42. Observe that there is a delay of about 6 ns in producing a change in the signal f from the time when the input signals, x_1 and x_2 , change their values. This delay is due to the propagation delays in the logic element and the wires in the FPGA device. You may also notice that a momentary change in the value of f , from 1 to 0 and back to 1, occurs at about 106-ns point in the simulation. This *glitch* is also due to the propagation delays in the FPGA device, because changes in x_1 and x_2 may not arrive at exactly the same time at the logic element that generates f .

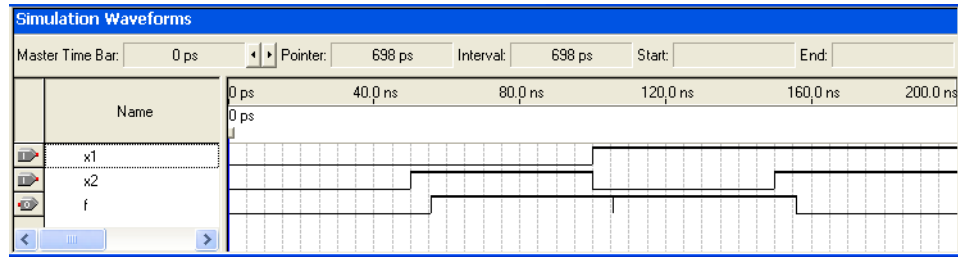


Figure 42. The result of timing simulation.

7 Programming and Configuring the FPGA Device

The FPGA device must be programmed and configured to implement the designed circuit. The required configuration file is generated by the Quartus II Compiler's Assembler module. Altera's DE2 board allows the configuration to be done in two different ways, known as JTAG and AS modes. The configuration data is transferred from the host computer (which runs the Quartus II software) to the board by means of a cable that connects a USB port on the host computer to the leftmost USB connector on the board. To use this connection, it is necessary to have the USB-Blaster driver installed. If this driver is not already installed, consult the tutorial *Getting Started with Altera's DE2 Board* for information about installing the driver. Before using the board, make sure that the USB cable is properly connected and turn on the power supply switch on the board.

In the JTAG mode, the configuration data is loaded directly into the FPGA device. The acronym JTAG stands for Joint Test Action Group. This group defined a simple way for testing digital circuits and loading data into them, which became an IEEE standard. If the FPGA is configured in this manner, it will retain its configuration as long as the power remains turned on. The configuration information is lost when the power is turned off. The second possibility is to use the Active Serial (AS) mode. In this case, a configuration device that includes some flash memory is used to store the configuration data. Quartus II software places the configuration data into the configuration device on the DE2 board. Then, this data is loaded into the FPGA upon power-up or reconfiguration. Thus, the FPGA need not be configured by the Quartus II software if the power is turned off and on. The choice between the two modes is made by the RUN/PROG switch on the DE2 board. The RUN position selects the JTAG mode, while the PROG position selects the AS mode.

7.1 JTAG Programming

The programming and configuration task is performed as follows. Flip the RUN/PROG switch into the RUN position. Select Tools > Programmer to reach the window in Figure 43. Here it is necessary to specify the programming hardware and the mode that should be used. If not already chosen by default, select JTAG in the Mode box. Also, if the USB-Blaster is not chosen by default, press the Hardware Setup... button and select the USB-Blaster in the window that pops up, as shown in Figure 44.

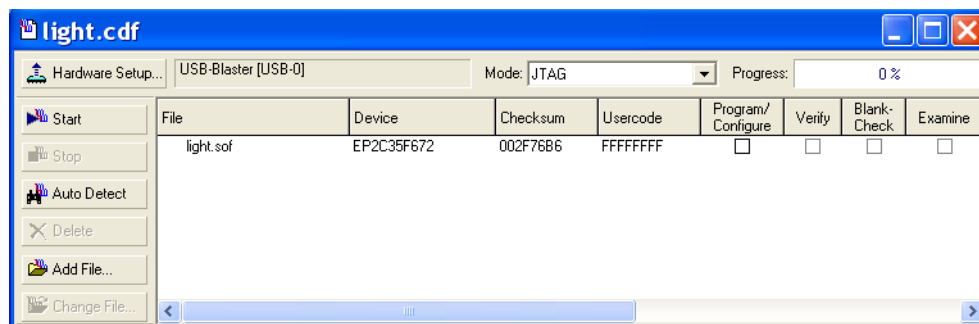


Figure 43. The Programmer window.

Observe that the configuration file *light.sof* is listed in the window in Figure 43. If the file is not already listed, then click **Add File** and select it. This is a binary file produced by the Compiler's Assembler module, which contains the data needed to configure the FPGA device. The extension *.sof* stands for SRAM Object File. Note also that the device selected is EP2C35F672, which is the FPGA device used on the DE2 board. Click on the **Program/Configure** check box, as shown in Figure 45.

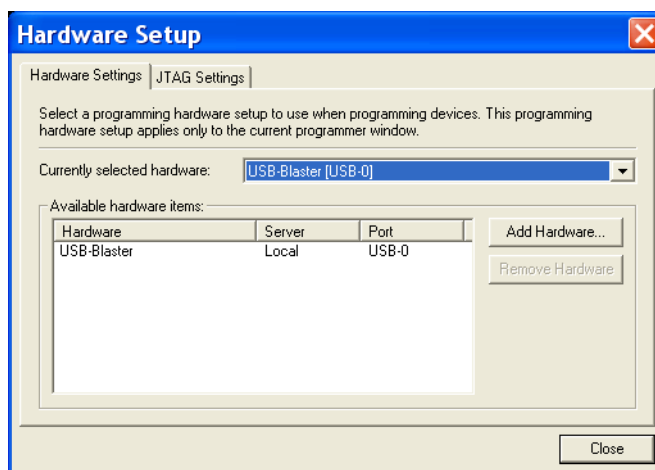


Figure 44. The Hardware Setup window.

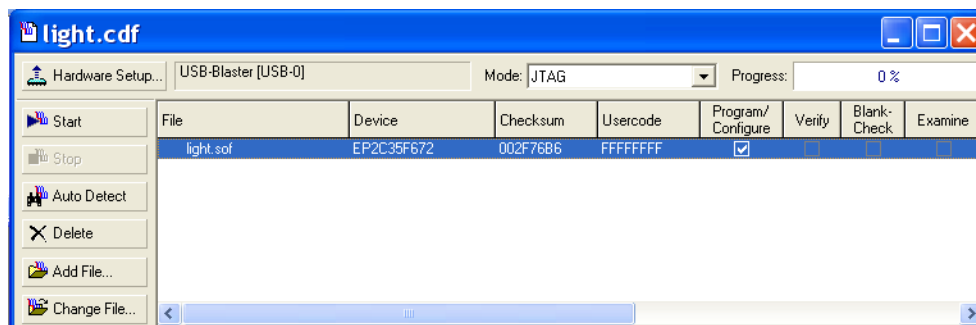


Figure 45. The updated Programmer window.

Now, press **Start** in the window in Figure 45. An LED on the board will light up when the configuration data has been downloaded successfully. If you see an error reported by Quartus II software indicating that programming failed, then check to ensure that the board is properly powered on.

7.2 Active Serial Mode Programming

In this case, the configuration data has to be loaded into the configuration device on the DE2 board, which is identified by the name EPCS16. To specify the required configuration device select **Assignments > Device**, which leads to the window in Figure 46. Click on the **Device & Pin Options** button to reach the window in Figure 47. Now, click on the **Configuration** tab to obtain the window in Figure 48. In the Configuration device box (which may be set to Auto) choose EPCS16 and click OK. Upon returning to the window in Figure 46, click OK. Recompile the designed circuit.

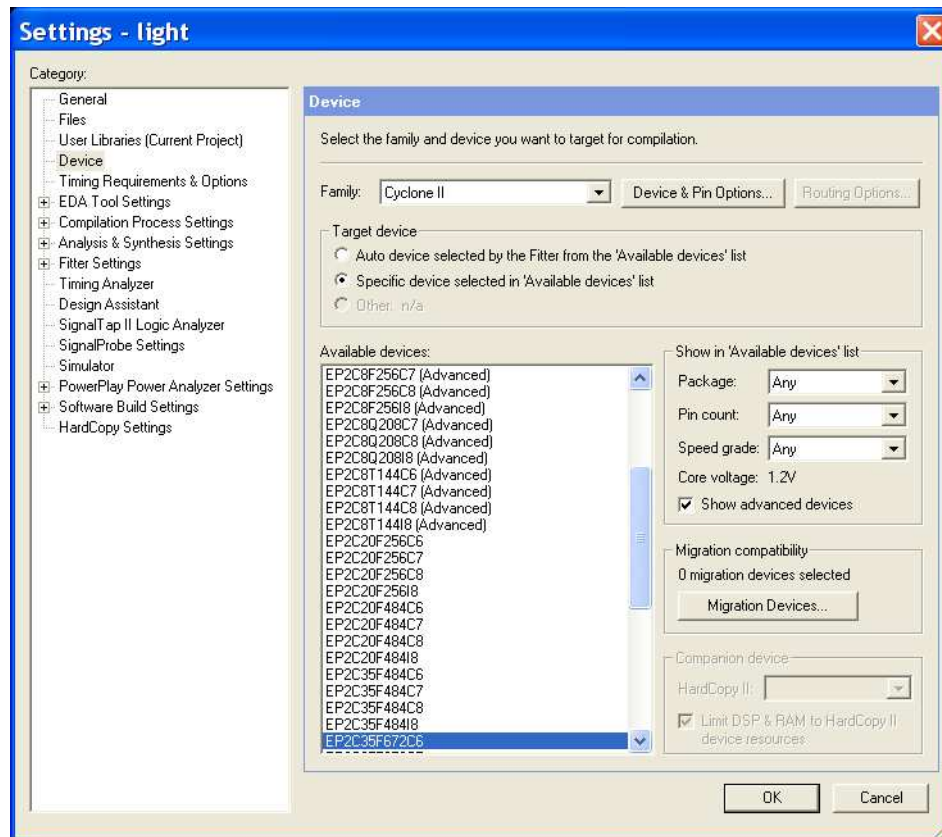


Figure 46. The Device Settings window.

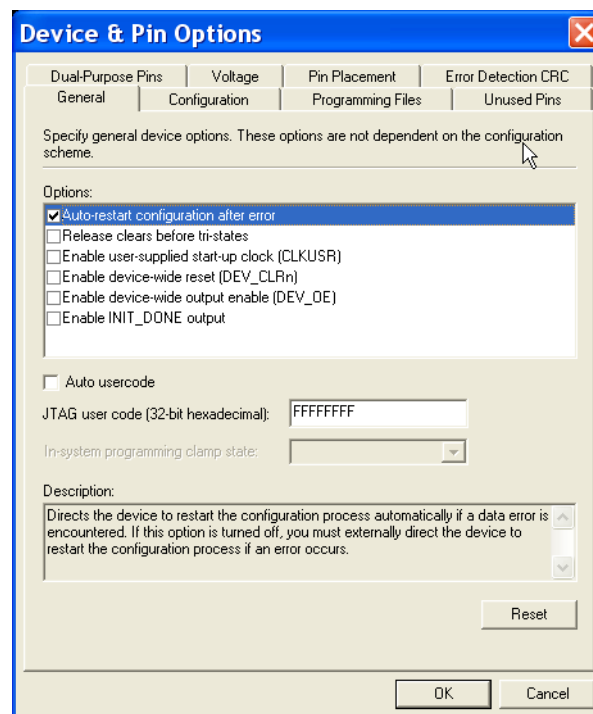


Figure 47. The Options window.

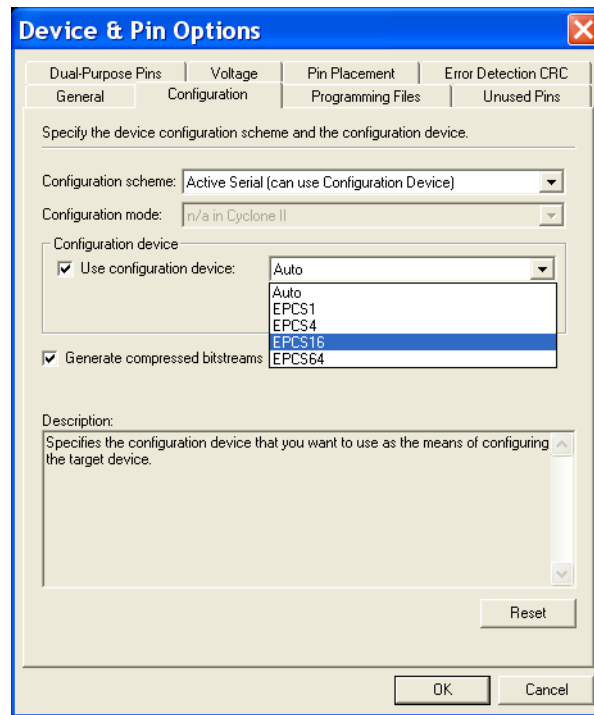


Figure 48. Specifying the configuration device.

The rest of the procedure is similar to the one described above for the JTAG mode. Select **Tools > Programmer** to reach the window in Figure 43. In the Mode box select **Active Serial Programming**. If you are changing the mode from the previously used JTAG mode, the pop-up box in Figure 49 will appear, asking if you want to clear all devices. Click **Yes**. Now, the Programmer window shown in Figure 50 will appear. Make sure that the Hardware Setup indicates the USB-Blaster. If the configuration file is not already listed in the window, press **Add File**. The pop-up box in Figure 51 will appear. Select the file *light.pof* in the directory *introtutorial* and click **Open**. As a result, the configuration file *light.pof* will be listed in the window. This is a binary file produced by the Compiler's Assembler module, which contains the data to be loaded into the EPCS16 configuration device. The extension *.pof* stands for Programmer Object File. Upon returning to the Programmer window, click on the **Program/Configure** check box, as shown in Figure 52.

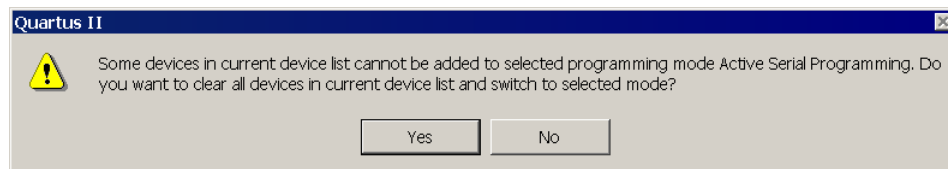


Figure 49. Clear the previously selected devices.

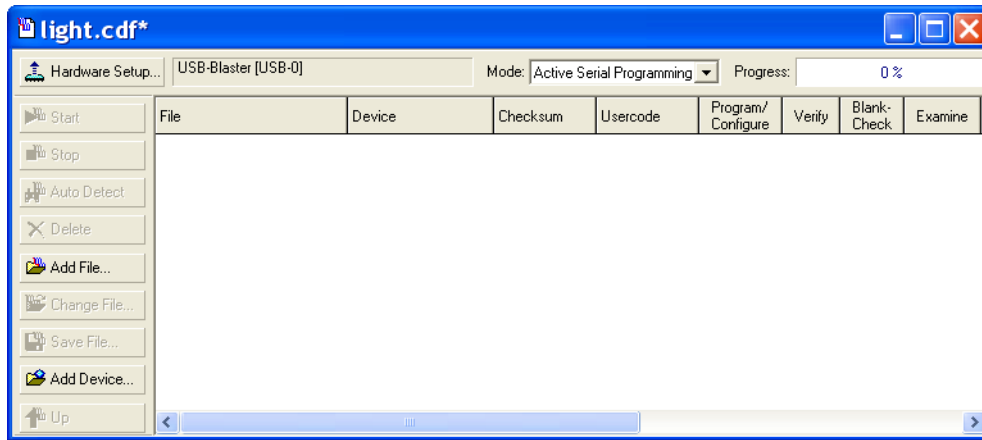


Figure 50. The Programmer window with Active Serial Programming selected.

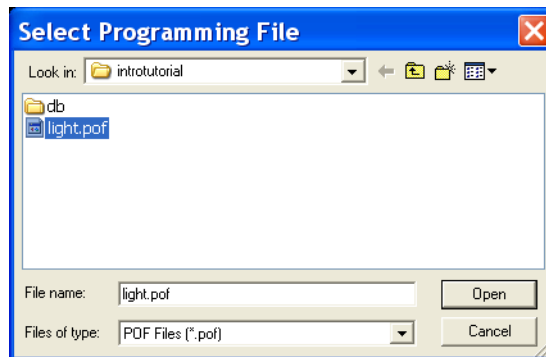


Figure 51. Choose the configuration file.

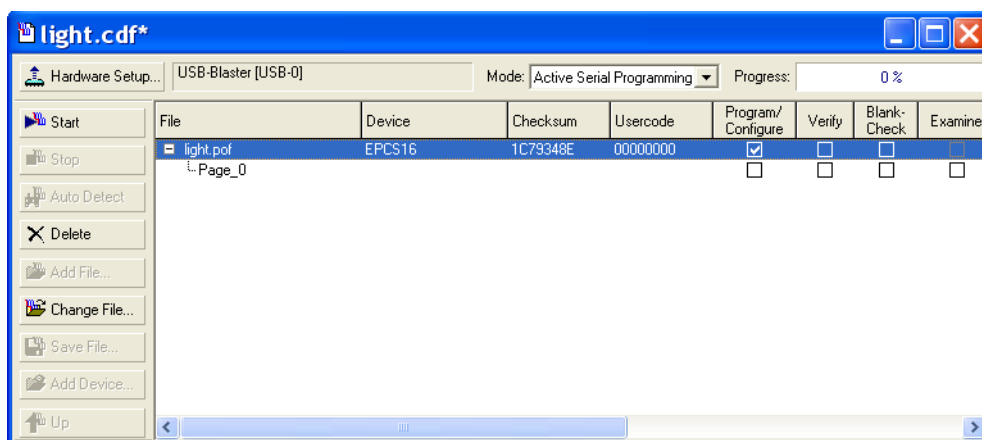


Figure 52. The updated Programmer window.

Flip the RUN/PROG switch on the DE2 board to the PROG position. Press Start in the window in Figure 52. An LED on the board will light up when the configuration data has been downloaded successfully. Also, the

Progress box in Figure 52 will indicate when the configuration and programming process is completed, as shown in Figure 53.

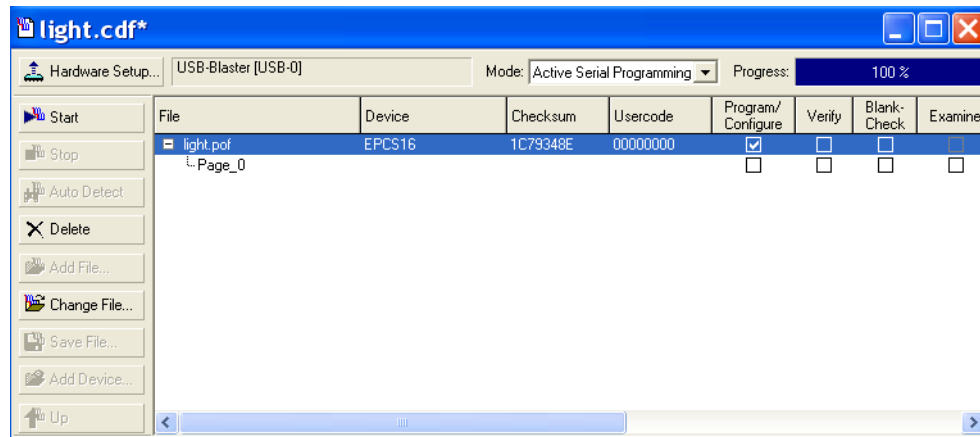


Figure 53. The Programmer window upon completion of programming.

8 Testing the Designed Circuit

Having downloaded the configuration data into the FPGA device, you can now test the implemented circuit. Flip the RUN/PROG switch to RUN position. Try all four valuations of the input variables x_1 and x_2 , by setting the corresponding states of the switches SW_1 and SW_0 . Verify that the circuit implements the truth table in Figure 12.

If you want to make changes in the designed circuit, first close the Programmer window. Then make the desired changes in the Block Diagram/Schematic file, compile the circuit, and program the board as explained above.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Using Library Modules in Verilog Designs

This tutorial explains how Altera's library modules can be included in Verilog-based designs, which are implemented by using the Quartus[®] II software.

Contents:

Example Circuit

Library of Parameterized Modules

Augmented Circuit with an LPM

Results for the Augmented Design

Practical designs often include commonly used circuit blocks such as adders, subtractors, multipliers, decoders, counters, and shifters. Altera provides efficient implementations of such blocks in the form of library modules that can be instantiated in Verilog designs. The compiler may recognize that a standard function specified in Verilog code can be realized using a library module, in which case it may automatically *infer* this module. However, many library modules provide functionality that is too complex to be recognized automatically by the compiler. These modules have to be instantiated in the design explicitly by the user.

Quartus[®] II software includes a *library of parameterized modules (LPM)*. The modules are general in structure and they are tailored to a specific application by specifying the values of general parameters.

Doing this tutorial, the reader will learn about:

- Library of parameterizes modules (LPMs)
- Configuring an LPM for use in a circuit
- Instantiating an LPM in a designed circuit

The detailed examples in the tutorial were obtained using the Quartus II version 5.0, but other versions of the software can also be used.

1 Example Circuit

As an example, we will use the adder/subtractor circuit shown in Figure 1. It can add, subtract, and accumulate n -bit numbers using the 2's complement number representation. The two primary inputs are numbers $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$, and the primary output is $Z = z_{n-1}z_{n-2} \cdots z_0$. Another input is the *AddSub* control signal which causes $Z = A + B$ to be performed when *AddSub* = 0 and $Z = A - B$ when *AddSub* = 1. A second control input, *Sel*, is used to select the accumulator mode of operation. If *Sel* = 0, the operation $Z = A \pm B$ is performed, but if *Sel* = 1, then B is added to or subtracted from the current value of Z . If the addition or subtraction operations result in arithmetic overflow, an output signal, *Overflow*, is asserted.

To make it easier to deal with asynchronous input signals, they are loaded into flip-flops on a positive edge of the clock. Thus, inputs A and B will be loaded into registers *Areg* and *Breg*, while *Sel* and *AddSub* will be loaded into flip-flops *SelR* and *AddSubR*, respectively. The adder/subtractor circuit places the result into register *Zreg*.

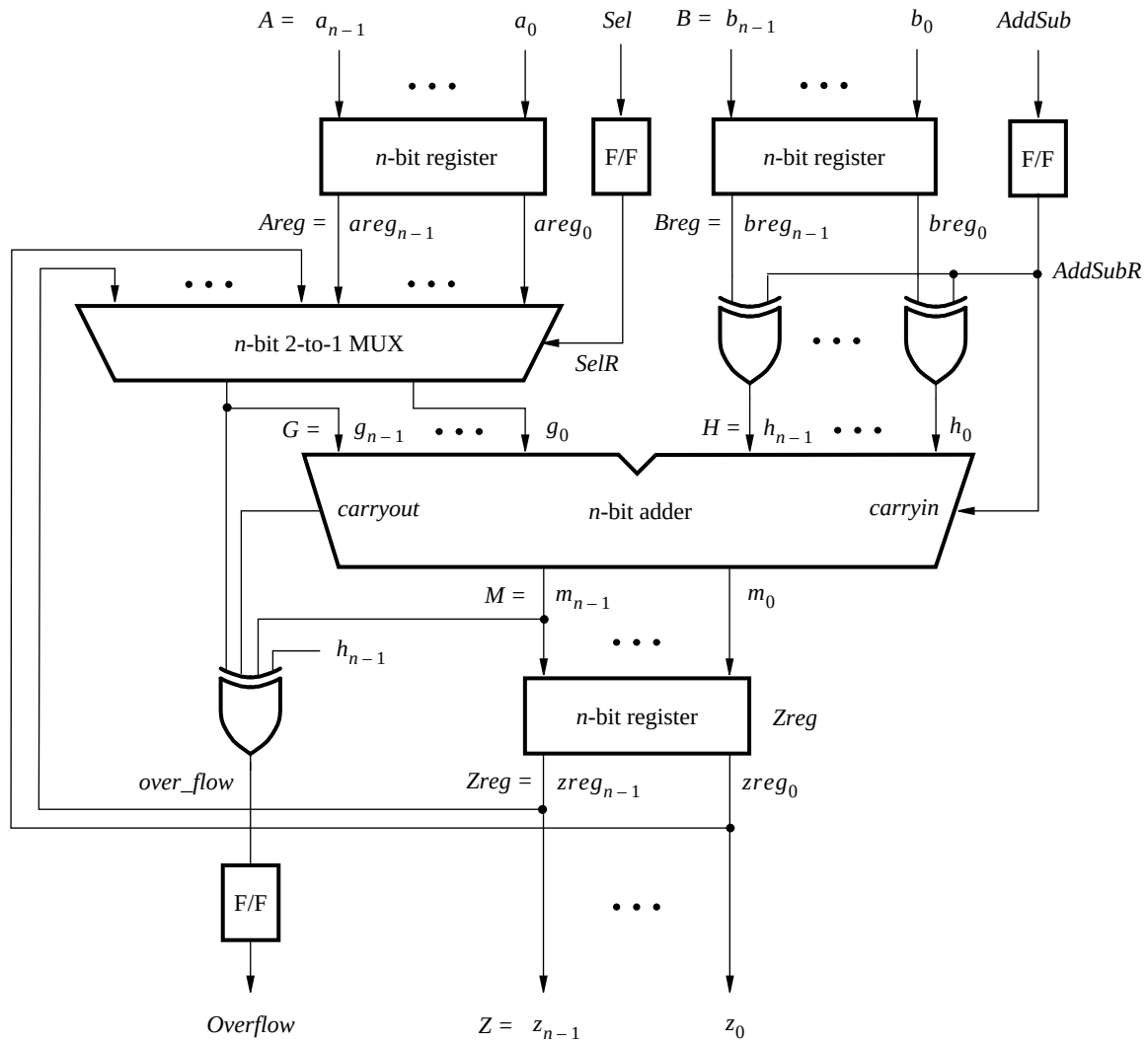


Figure 1. The adder/subtractor circuit.

The required circuit is described by the Verilog code in Figure 2. For our example, we use a 16-bit circuit as specified by $n = 16$. Implement this circuit as follows:

- Create a project *addersubtractor*.
- Include a file *addersubtractor.v*, which corresponds to Figure 2, in the project. For convenience, this file is provided in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages.
- Choose the Cyclone II EP2C35F672C6 device, which is the FPGA chip on Altera's DE2 board.
- Compile the design.
- Simulate the design by applying some typical inputs.

```

// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, H, M, Z;
    wire carryout, over_flow;

// Define combinational logic circuit
    assign H = Breg ^ {n{AddSubR}};
    mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
    adderk nbit_adder (AddSubR, G, H, M, carryout);
    defparam nbit_adder.k = n;
    assign over_flow = carryout ^ G[n-1] ^ H[n-1] ^ M[n-1];
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
            begin
                Areg <= 0; Breg <= 0; Zreg <= 0;
                SelR <= 0; AddSubR <= 0; Overflow <= 0;
            end
        else
            begin
                Areg <= A; Breg <= B; Zreg <= M;
                SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
            end
    endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Selm, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Selm;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Selm)
        if (Selm == 0) F = V;
        else F = W;
endmodule

... continued in Part b

```

Figure 2. Verilog code for the circuit in Figure 1 (Part a).

```

// k-bit adder
module adderk (carryin, X, Y, S, carryout);
    parameter k = 8;
    input [k-1:0] X, Y;
    input carryin;
    output [k-1:0] S;
    output carryout;
    reg [k-1:0] S;
    reg carryout;

    always @(X or Y or carryin)
        {carryout, S} = X + Y + carryin;
endmodule

```

Figure 2. Verilog code for the circuit in Figure 1 (Part *b*).

2 Library of Parameterized Modules

The LPMs in the library of parameterized modules are general in structure and they can be configured to suit a specific application by specifying the values of various parameters. Select **Help > Megafunctions/LPM** to see a listing of the available LPMs. One of them is an adder/subtractor module called *lpm_add_sub megafunction*. Select this module to see its description. The module has a number of inputs and outputs, some of which may be omitted in a given application. Several parameters can be defined to specify a particular mode of operation. For example, the number of bits in the operands is specified in the parameter `LPM_WIDTH`. The `LPM_REPRESENTATION` parameter specifies whether the operands are to be interpreted as signed or unsigned numbers, and so on. Templates on how an LPM can be instantiated in a hardware description language are given in the description of the module. Using these templates is somewhat cumbersome, so Quartus II software provides a wizard that makes the instantiation of LPMs easy.

We will use the *lpm_add_sub* module to simplify our adder/subtractor circuit defined in Figures 1 and 2. The augmented circuit is given in Figure 3. The *lpm_add_sub* module, instantiated under the name *megaddsub*, replaces the adder circuit as well as the XOR gates that provide the input *H* to the adder. Since arithmetic overflow is one of the outputs that the LPM provides, it is not necessary to generate this output with a separate XOR gate.

To implement this adder/subtractor circuit, create a new directory named *tutorial_lpm*, and then create a project *addersubtractor2*. Choose the same Cyclone II EP2C35F672C6 device, to allow a direct comparison of implemented designs.

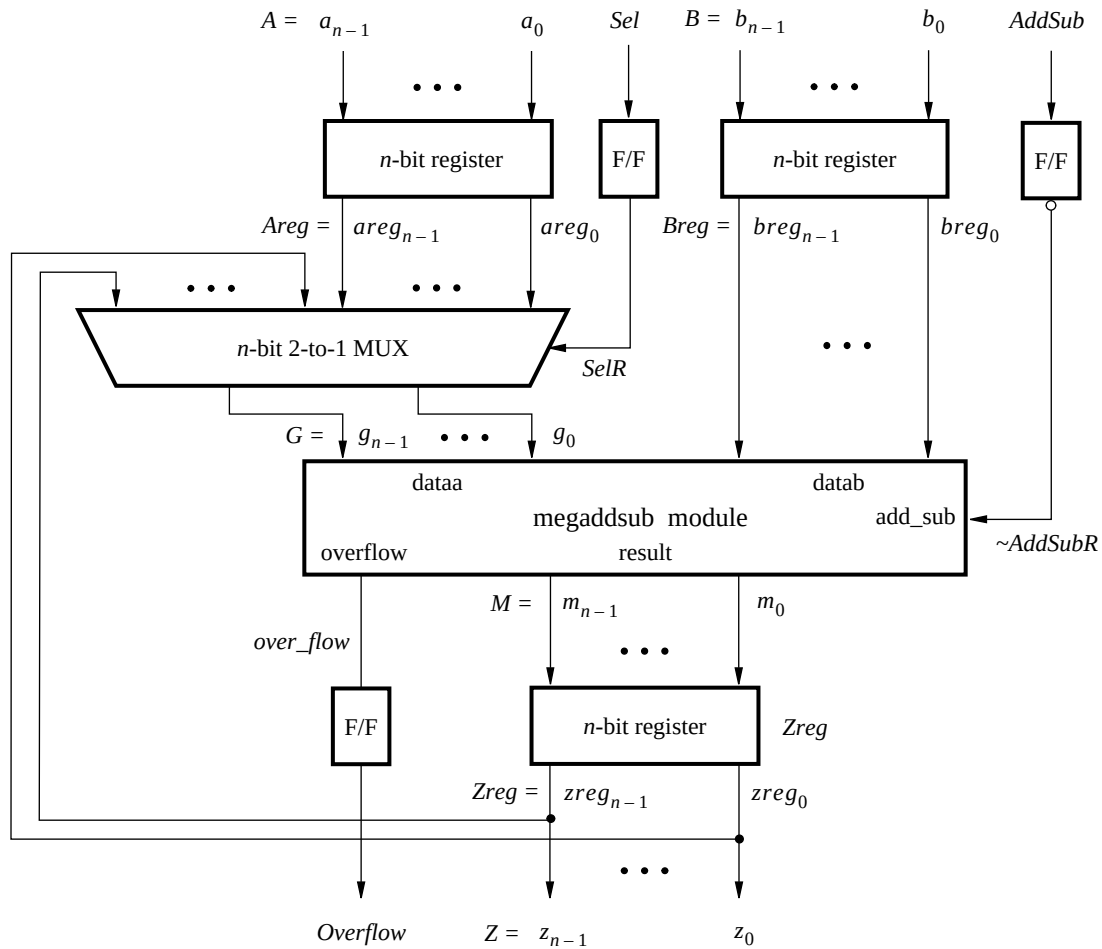


Figure 3. The augmented adder/subtractor circuit.

The new design will include the desired LPM subcircuit specified as a Verilog module that will be instantiated in the top-level Verilog design module. The Verilog module for the LPM subcircuit is generated by using a wizard as follows:

1. Select Tools > MegaWizard Plug-in Manager, which leads to a sequence of seven pop-up boxes in which the user can specify the details of the desired LPM.
2. In the box shown in Figure 4 indicate Create a new custom megafunction variation and click Next.

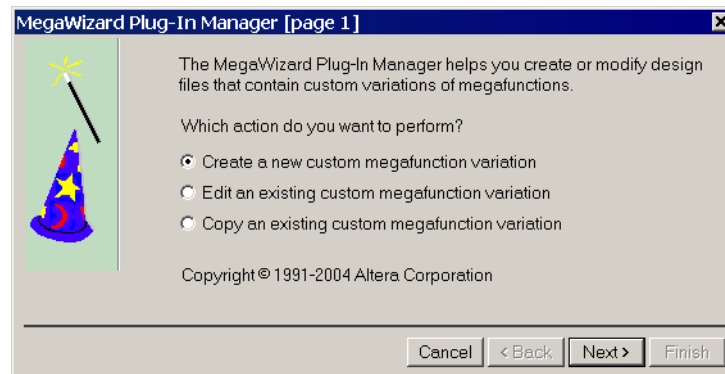


Figure 4. Choose to define an LPM.

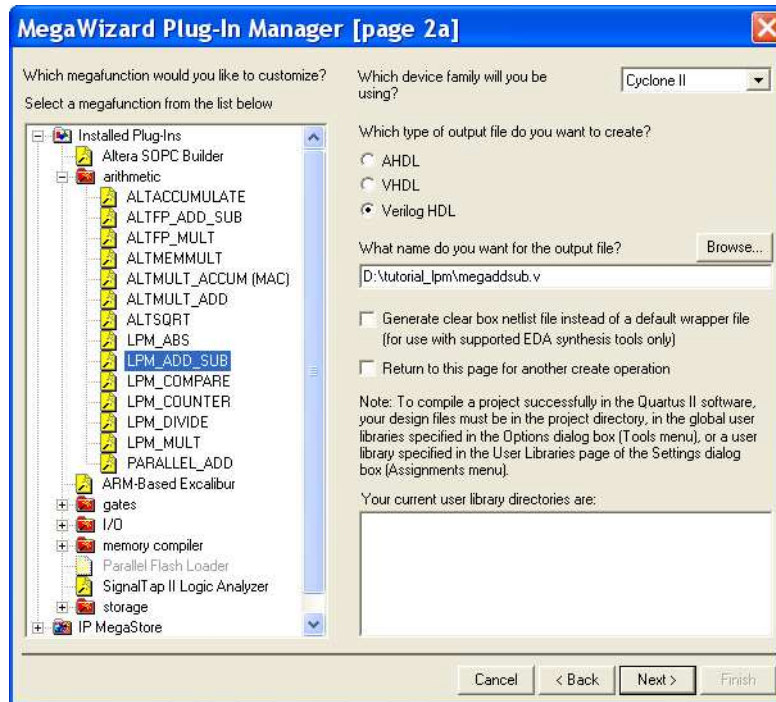


Figure 5. Choose an LPM from the available library.

3. The box in Figure 5 provides a list of the available LPMs. Expand the “arithmetic” sublist and select LPM_ADD_SUB. Choose Verilog HDL as the type of output file that should be created. The output file must be given a name; choose the name *megaddsub.v* and indicate that the file should be placed in the directory *tutorial_lpm* as shown in the figure. Press Next.

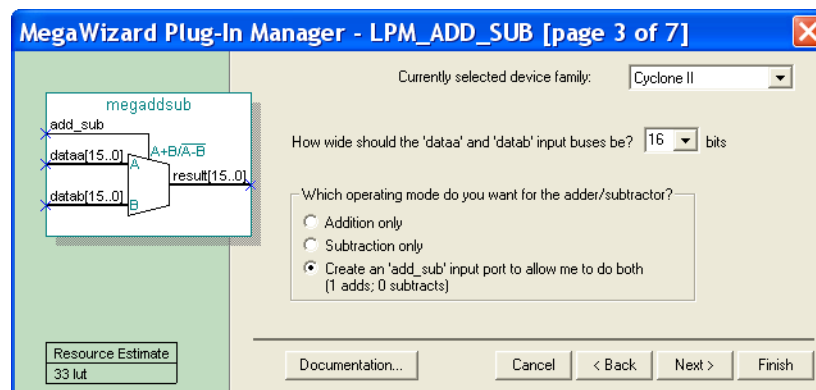


Figure 6. Specify the size of data inputs.

4. In the box in Figure 6 specify that the width of the data inputs is 16 bits. Also, specify the operating mode in which one of the ports allows performing both addition and subtraction of the input operand, under the control of the *add_sub* input. A symbol for the resulting LPM is shown in the top left corner. Note that if *add_sub* = 1 then *result* = $A + B$; otherwise, *result* = $A - B$. This interpretation of the control input and the operation performed is different from our original design in Figures 1 and 2, which we have to account for in the modified design. Observe that we have included this change in the circuit in Figure 3. Click Next.

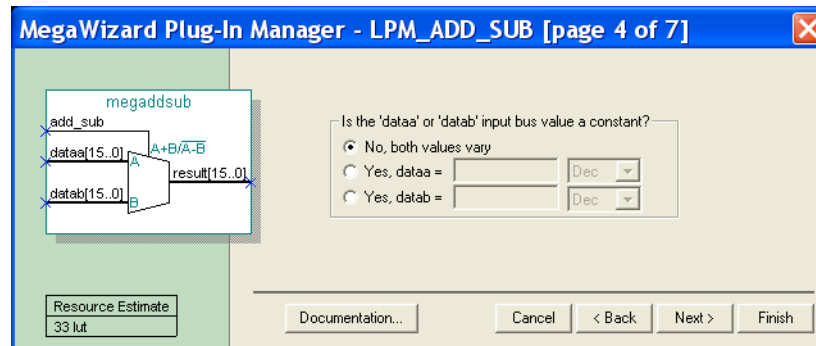


Figure 7. Further specification of inputs.

5. In the box in Figure 7, specify that the values of both inputs may vary and click Next.

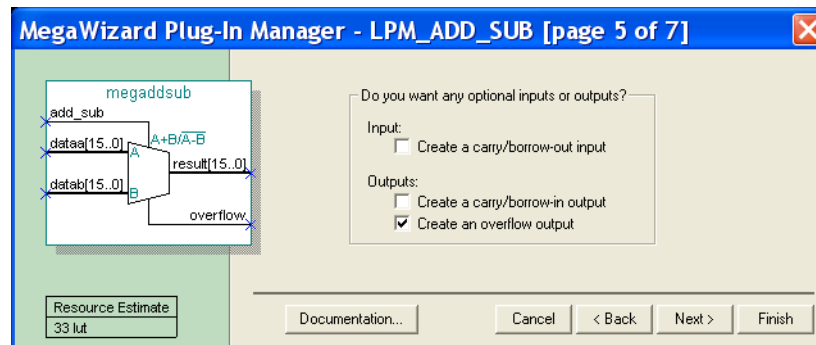


Figure 8. Specify the Overflow output.

6. The box in Figure 8 allows the designer to indicate optional inputs and outputs that may be specified. Since we need the overflow signal, make the Create an overflow output choice and press Next.

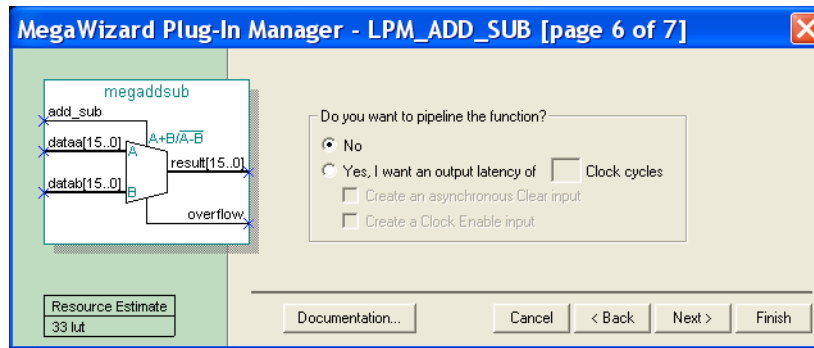


Figure 9. Refuse the pipelining option.

7. In the box in Figure 9 say NO to the pipelining option and click Next.
8. Figure 10 gives a summary which shows the files that the wizard will create. Press Finish to complete the process.

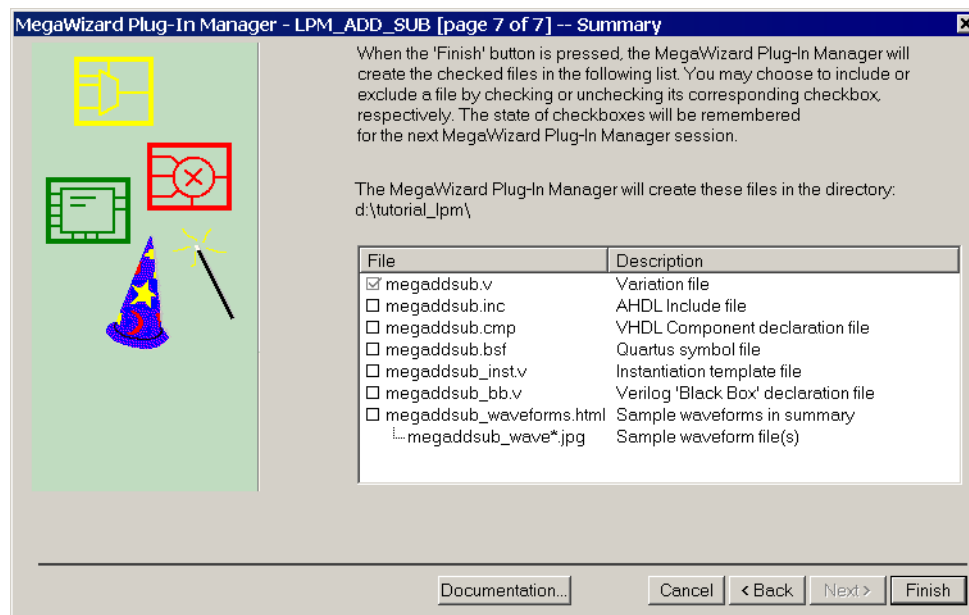


Figure 10. Files created by the wizard.

3 Augmented Circuit with an LPM

We will use the file *megaddsub.v* in our modified design. Figure 11 depicts the Verilog code in this file; note that we have not shown the comments in order to keep the figure small.

```

// Adder/subtractor module created by the MegaWizard
module megaddsub (
    add_sub,
    dataa,
    datab,
    result,
    overflow);

    input add_sub;
    input [15:0] dataa;
    input [15:0] datab;
    output [15:0] result;
    output overflow;
    wire sub_wire0;
    wire [15:0] sub_wire1;
    wire overflow = sub_wire0;
    wire [15:0] result = sub_wire1[15:0];

    lpm_add_sub lpm_add_sub_component (
        .dataa (dataa),
        .add_sub (add_sub),
        .datab (datab),
        .overflow (sub_wire0),
        .result (sub_wire1));
    defparam
        lpm_add_sub_component.lpm_width = 16,
        lpm_add_sub_component.lpm_direction = "UNUSED",
        lpm_add_sub_component.lpm_type = "LPM_ADD_SUB",
        lpm_add_sub_component.lpm_hint = "ONE_INPUT_IS_CONSTANT=NO";
endmodule

```

Figure 11. Verilog code for the ADD_SUB LPM.

The modified Verilog code for the adder/subtractor design is given in Figure 12. Put this code into a file *tutorial_lpm\addersubtractor2.v*. For convenience, the required file *addersubtractor2.v* is provided in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages. The differences between this code and Figure 2 are:

- The **assign** statements that define the *overflow* signal and the XOR gates (along with the signal defined as **wire H**) are no longer needed.
- The *adderk* instance of the adder circuit is replaced by *megaddsub*. Note that the *dataa* and *datab* inputs shown in Figure 6 are driven by the *G* and *Breg* vectors, respectively. Also, the inverted version of the *AddSubR* signal is specified to conform with the usage of this control signal in the LPM.
- The *adderk* module is deleted from the code.

```

// Top-level module
module addersubtractor2 (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, M, Z;
    wire over_flow;

// Define combinational logic circuit
    mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
    megaddsub nbit_adder (~AddSubR, G, Breg, M, over_flow);
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
            begin
                Areg <= 0; Breg <= 0; Zreg <= 0;
                SelR <= 0; AddSubR <= 0; Overflow <= 0;
            end
        else
            begin
                Areg <= A; Breg <= B; Zreg <= M;
                SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
            end
    endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Selm, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Selm;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Selm)
        if (Selm == 0) F = V;
        else F = W;
endmodule

```

Figure 12. Verilog code for the circuit in Figure 3.

To include the *megaddsub.v* file in the project, select **Project > Add/Remove Files in Project** to reach the window in Figure 13. The file *addersubtractor2.v* should already be listed as being included in the project. Browse for the other files by clicking the button **File name: ...** to reach the window in Figure 14. Select the file *megaddsub.v* and click **Open**, which returns to the window in Figure 13. Click **Add** to include the file and then click **OK**. Now, the modified design can be compiled and simulated in the usual way.

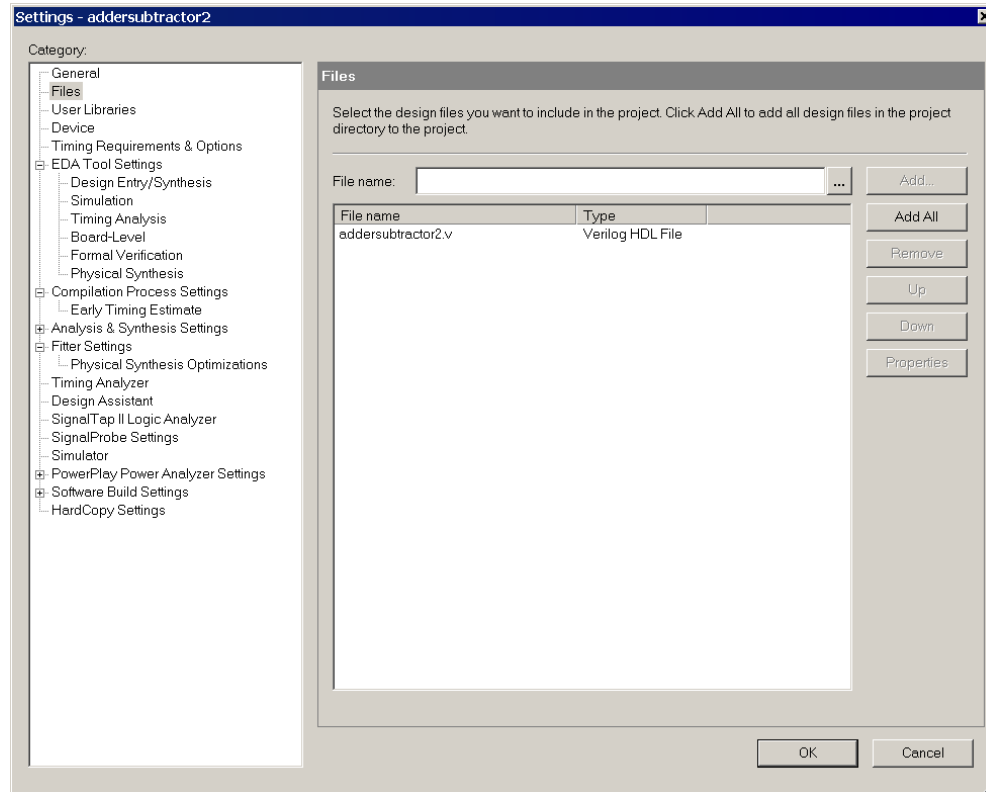


Figure 13. Inclusion of the new file in the project.

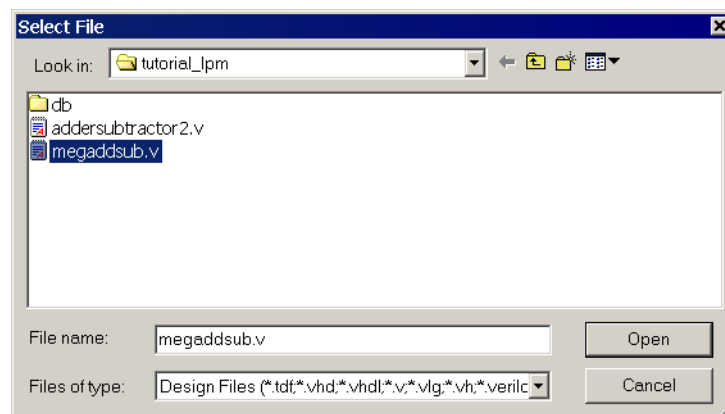
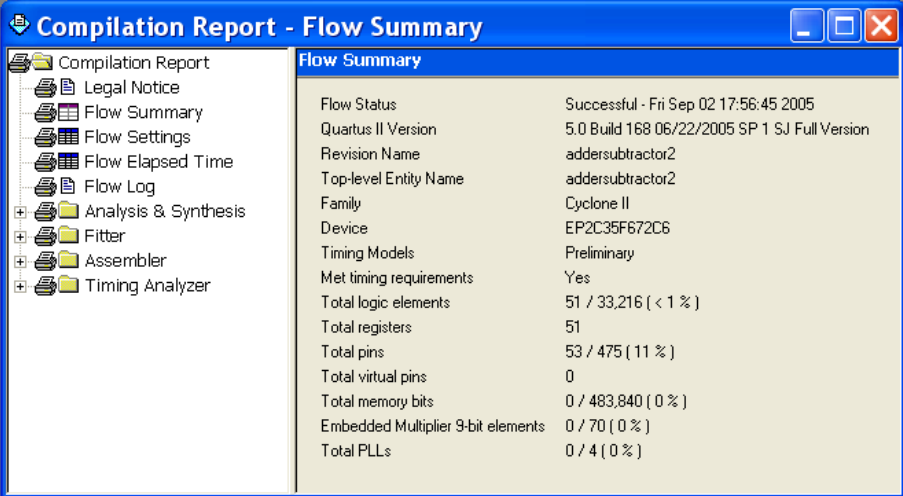


Figure 14. Specify the *megaddsub.v* file.

4 Results for the Augmented Design

Compile the design and look at the summary, which is depicted in Figure 15. Observe that the modified design is implemented in 51 logic elements, rather than 52 obtained when using the code in Figure 2. The reduction in the number of logic elements is small because our example is rather simple. In more complex designs the advantage of using LPMs is likely to be more pronounced. The reason is that the LPMs implement the required logic more efficiently than what the compiler can do from simple Verilog code, such as the code in Figure 2. The user should consider using an LPM whenever a suitable one exists.



Flow Summary	
Flow Status	Successful - Fri Sep 02 17:56:45 2005
Quartus II Version	5.0 Build 168 06/22/2005 SP 1 SJ Full Version
Revision Name	addersubtractor2
Top-level Entity Name	addersubtractor2
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Preliminary
Met timing requirements	Yes
Total logic elements	51 / 33,216 (< 1 %)
Total registers	51
Total pins	53 / 475 (11 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 15. Compilation Results for the Augmented Circuit.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Timing Considerations with Verilog-Based Designs

This tutorial describes how Altera's Quartus[®] II software deals with the timing issues in designs based on the Verilog hardware description language. It discusses the various timing parameters and explains how specific timing constraints may be set by the user.

Contents:

- Example Circuit
- Timing Analyzer Report
- Specifying the Timing Constraints
- Timing Simulation

Quartus II software includes a Timing Analyzer module which performs a detailed analysis of all timing delays for a circuit that is compiled for implementation in an FPGA chip. This tutorial discusses the types of analyses performed and shows how particular timing requirements may be specified by the user. The discussion assumes that the reader is familiar with the basic operation of Quartus II software, as may be learned from an introductory tutorial.

Doing this tutorial, the reader will learn about:

- Parameters evaluated by the Timing Analyzer
- Specifying the desired values of timing parameters
- Using timing simulation

The timing results shown in the examples in this tutorial were obtained using Quartus II version 5.0, but other versions of the software can also be used.

1 Example Circuit

Timing issues are most important in circuits that involve long paths through combinational logic elements with registers at inputs and outputs of these paths. As an example, we will use the adder/subtractor circuit shown in Figure 1. It can add, subtract, and accumulate n -bit numbers using the 2's complement number representation. The two primary inputs are numbers $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$, and the primary output is $Z = z_{n-1}z_{n-2} \cdots z_0$. Another input is the *AddSub* control signal which causes $Z = A + B$ to be performed when *AddSub* = 0 and $Z = A - B$ when *AddSub* = 1. A second control input, *Sel*, is used to select the accumulator mode of operation. If *Sel* = 0, the operation $Z = A \pm B$ is performed, but if *Sel* = 1, then B is added to or subtracted from the current value of Z . If the addition or subtraction operations result in arithmetic overflow, an output signal, *Overflow*, is asserted.

To make it easier to deal with asynchronous input signals, they are loaded into flip-flops on a positive edge of the clock. Thus, inputs A and B will be loaded into registers *Areg* and *Breg*, while *Sel* and *AddSub* will be loaded into flip-flops *SelR* and *AddSubR*, respectively. The adder/subtractor circuit places the result into register *Zreg*.

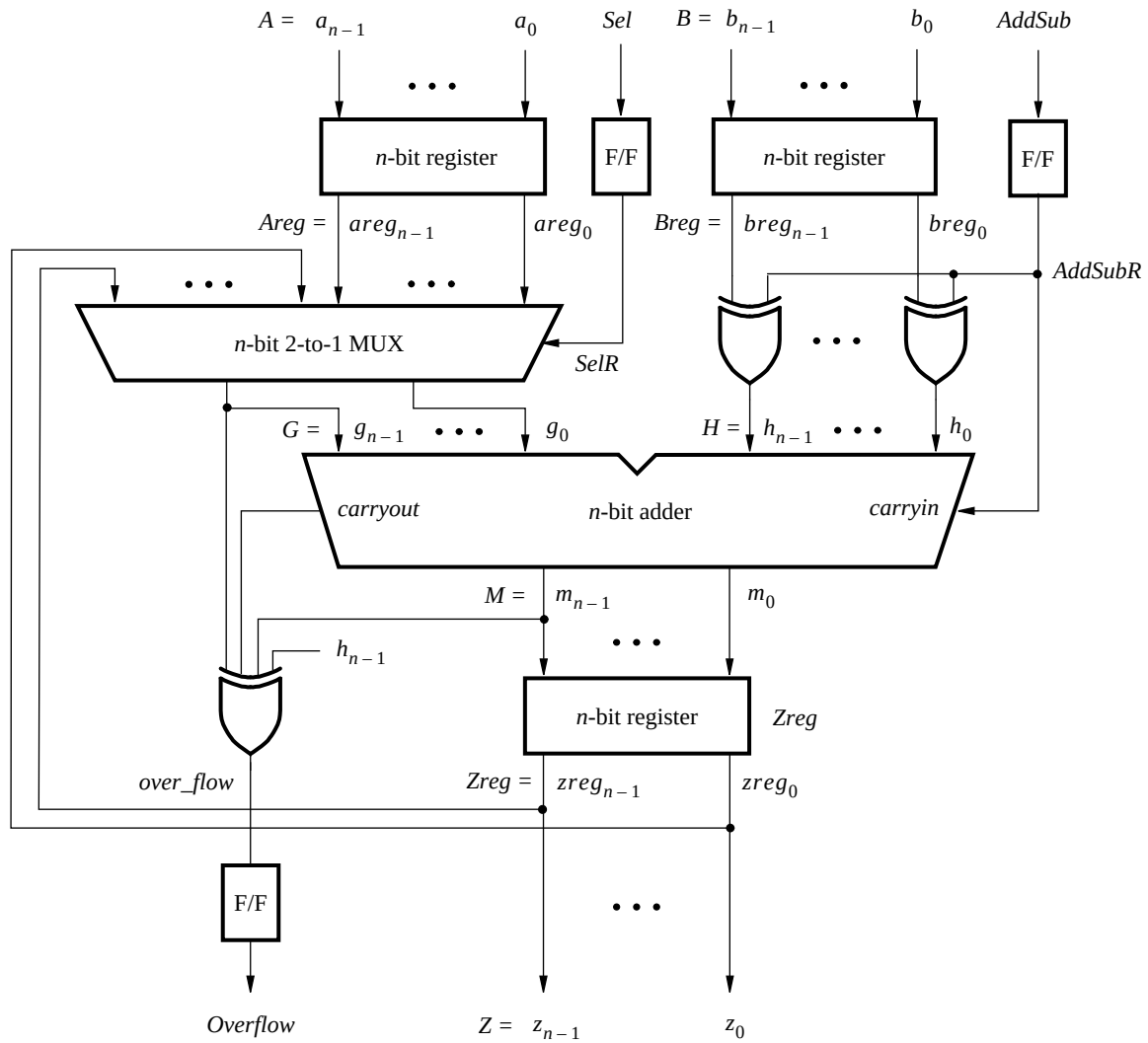


Figure 1. The adder/subtractor circuit.

The required circuit is described by the Verilog code in Figure 2. For our example, we use a 16-bit circuit as specified by $n = 16$. Implement this circuit as follows:

- Create a project *addersubtractor*.
- Include a file *addersubtractor.v*, which corresponds to Figure 2, in the project. For convenience, this file is provided in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages.
- Choose the Cyclone II EP2C35F672C6 device, which is the FPGA chip on Altera's DE2 board.
- Compile the design.

```

// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, H, M, Z;
    wire carryout, over_flow;

// Define combinational logic circuit
    assign H = Breg ^ {n{AddSubR}};
    mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
    adderk nbit_adder (AddSubR, G, H, M, carryout);
    defparam nbit_adder.k = n;
    assign over_flow = carryout ^ G[n-1] ^ H[n-1] ^ M[n-1];
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
            begin
                Areg <= 0; Breg <= 0; Zreg <= 0;
                SelR <= 0; AddSubR <= 0; Overflow <= 0;
            end
        else
            begin
                Areg <= A; Breg <= B; Zreg <= M;
                SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
            end
    endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Selm, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Selm;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Selm)
        if (Selm == 0) F = V;
        else F = W;
endmodule

... continued in Part b

```

Figure 2. Verilog code for the circuit in Figure 1 (Part a).

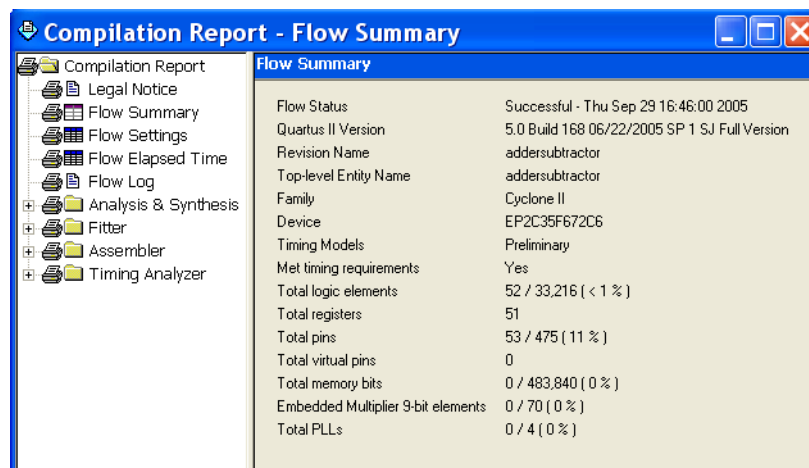
```
// k-bit adder
module adderk (carryin, X, Y, S, carryout);
    parameter k = 8;
    input [k-1:0] X, Y;
    input carryin;
    output [k-1:0] S;
    output carryout;
    reg [k-1:0] S;
    reg carryout;

    always @(X or Y or carryin)
        {carryout, S} = X + Y + carryin;
endmodule
```

Figure 2. Verilog code for the circuit in Figure 1 (Part b).

2 Timing Analyzer Report

Successful compilation of our circuit generates the Compilation Report in Figure 3. This report provides a lot of useful information. It shows the number of logic elements, flip-flops (called registers), and pins needed to implement the circuit. It gives detailed information produced by the Synthesis and Fitter modules. It also indicates the speed of the implemented circuit. A good measure of the speed is the maximum frequency at which the circuit can be clocked, referred to as *f_{max}*. This measure depends on the longest delay along any path, called the *critical path*, between two registers clocked by the same clock. Quartus II software performs a timing analysis to determine the expected performance of the circuit. It evaluates several parameters, which are listed in the Timing Analyzer section of the Compilation Report. Click on the small + symbol next to Timing Analyzer to expand this section of the report, and then click on the Timing Analyzer item Summary which displays the table in Figure 4. The last entry in the table shows that the maximum frequency for our circuit implemented on the specified chip is 214.27 MHz. You may get a different value of *f_{max}*, dependent on the specific version of Quartus II software that you are using. To see the paths in the circuit that limit the *f_{max}*, click on the Timing Analyzer item Clock Setup: 'Clock' in Figure 4 to obtain the display in Figure 5. This table shows that the critical path begins at the flip-flop *AddSubR* and ends at the flip-flop *Overflow*.



Flow Status	Successful - Thu Sep 29 16:46:00 2005
Quartus II Version	5.0 Build 168 06/22/2005 SP 1 SJ Full Version
Revision Name	addersubtractor
Top-level Entity Name	addersubtractor
Family	Cyclone II
Device	EP2K35F672C6
Timing Models	Preliminary
Met timing requirements	Yes
Total logic elements	52 / 33,216 (< 1 %)
Total registers	51
Total pins	53 / 475 (11 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 3. The Compilation Report.

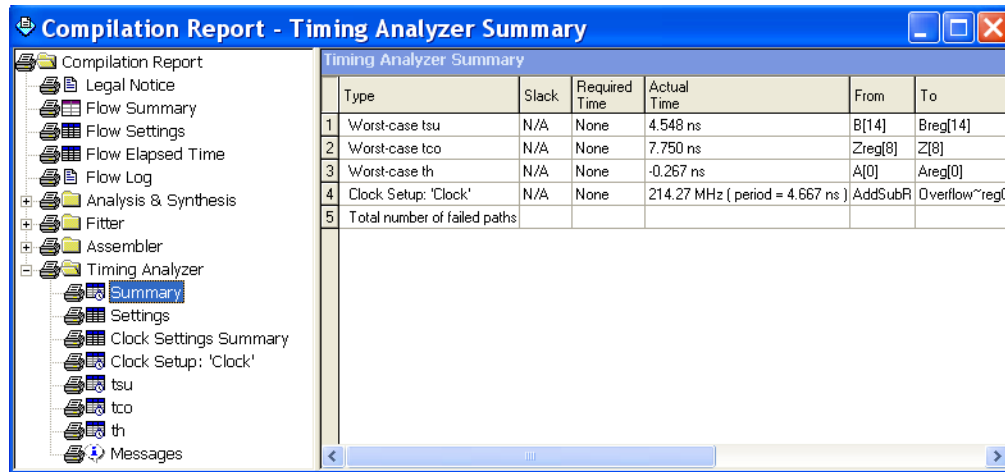


Figure 4. The Timing Analyzer Summary.

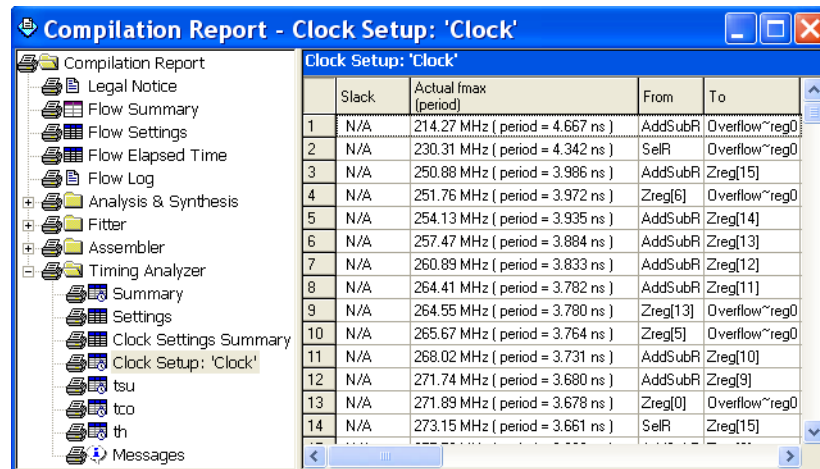


Figure 5. Critical paths.

The table in Figure 4 also shows other timing results. While f_{max} is a function of the longest propagation delay between two registers in the circuit, it does not indicate the delays with which output signals appear at the pins of the chip. The time elapsed from an active edge of the clock signal at the clock source until a corresponding output signal is produced (from a flip-flop) at an output pin is denoted as the t_{co} delay at that pin. In the worst case, the t_{co} in our circuit is 7.750 ns. Click on t_{co} in the Timing Analyzer section to view the table given in Figure 6. The first entry in the table shows that it takes 7.750 ns from when an active clock edge occurs until a signal propagates from bit 8 in register $Zreg$ to the output pin z_8 . The other two parameters given in Figure 4 are setup time, tsu , and hold time, th .

	Slack	Required tco	Actual tco	From	To	From Clock
1	N/A	None	7.750 ns	Zreg[8]	Z[8]	Clock
2	N/A	None	7.503 ns	Overflow~reg0	Overflow	Clock
3	N/A	None	7.456 ns	Zreg[12]	Z[12]	Clock
4	N/A	None	7.411 ns	Zreg[3]	Z[3]	Clock
5	N/A	None	7.328 ns	Zreg[2]	Z[2]	Clock
6	N/A	None	7.311 ns	Zreg[9]	Z[9]	Clock
7	N/A	None	7.196 ns	Zreg[6]	Z[6]	Clock
8	N/A	None	7.173 ns	Zreg[0]	Z[0]	Clock
9	N/A	None	7.093 ns	Zreg[11]	Z[11]	Clock
10	N/A	None	7.088 ns	Zreg[5]	Z[5]	Clock
11	N/A	None	7.072 ns	Zreg[10]	Z[10]	Clock
12	N/A	None	7.068 ns	Zreg[14]	Z[14]	Clock
13	N/A	None	7.046 ns	Zreg[13]	Z[13]	Clock
14	N/A	None	7.046 ns	Zreg[4]	Z[4]	Clock
15	N/A	None	6.801 ns	Zreg[1]	Z[1]	Clock
16	N/A	None	6.744 ns	Zreg[15]	Z[15]	Clock
17	N/A	None	6.547 ns	Zreg[7]	Z[7]	Clock

Figure 6. The *tco* delays.

3 Specifying Timing Constraints

So far we have compiled our Verilog code without indicating to the Quartus II software the required speed performance of the circuit. In the absence of such timing constraints the Quartus II software implements a designed circuit in a good but not necessarily the best way in order to keep the compilation time short. If the result does not meet the user's expectations, it is possible to specify certain timing constraints that should be met. For example, suppose that we want our example circuit to operate at a clock frequency of at least 250 MHz, rather than the 214.27 MHz as indicated by the value of *fmax* in Figure 4. To see if this can be achieved we can set the *fmax* constraint as follows:

1. Select Assignments > Timing Settings to reach the Timing Requirements & Options window in Figure 7. In this window it is possible to specify the requirements for a number of different parameters.
2. In the box Clock Settings specify that the required value of *fmax* is 250 MHz. Click OK.
3. Recompile the circuit.
4. Open the Timing Analyzer Summary to see that the new *fmax* is 263.02 MHz, as indicated in Figure 8. You may get a slightly different result depending on the version of the Quartus II software used.

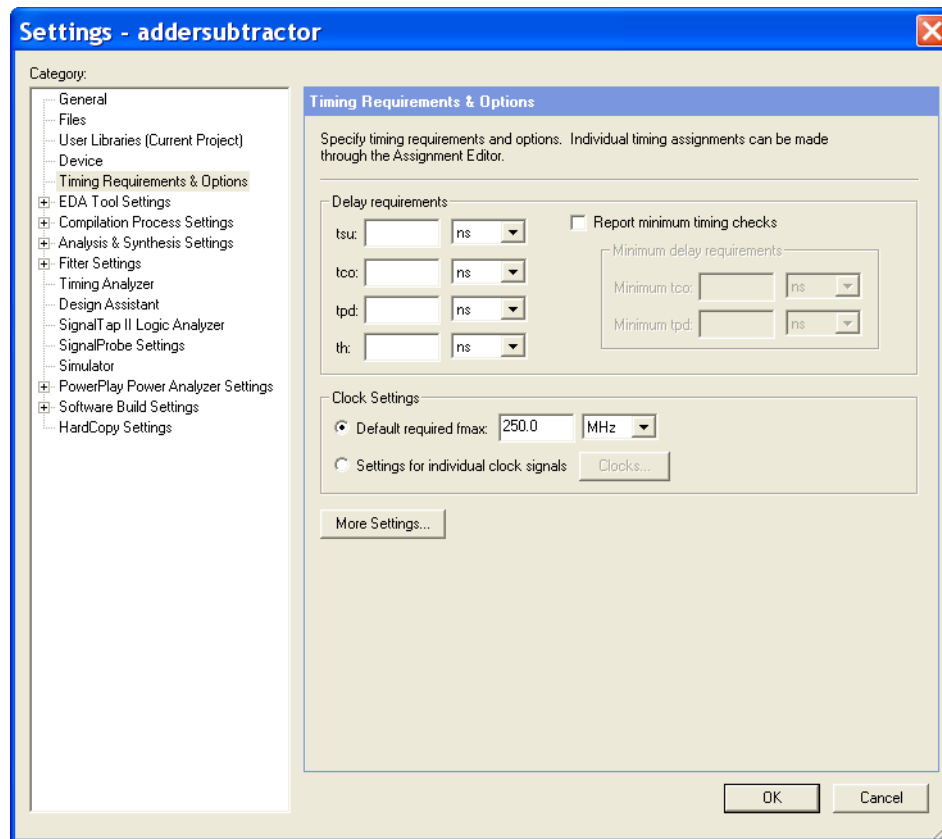


Figure 7. Specify the timing constraints in the Settings window.

Type	Slack	Required Time	Actual Time	From	To
1 Worst-case tsu	N/A	None	4.191 ns	A[15]	Areg[15]
2 Worst-case tco	N/A	None	7.181 ns	Zreg[8]	Z[8]
3 Worst-case th	N/A	None	-0.558 ns	Sel	SelR
4 Clock Setup: 'Clock'	0.198 ns	250.00 MHz (period = 4.000 ns)	263.02 MHz (period = 3.802 ns)	AddSubR	Overflow~reg0
5 Clock Hold: 'Clock'	1.175 ns	250.00 MHz (period = 4.000 ns)	N/A	Areg[0]	Zreg[0]
6 Total number of failed paths					

Figure 8. New timing results.

If the specified constraint is too high, the Quartus II compiler will not be able to satisfy it. For example, set the *fmax* constraint to 300 MHz and recompile the circuit. Now, the Timing Analyzer Summary will show that this constraint cannot be met, as seen in Figure 9.

Type	Slack	Required Time	Actual Time	From	To
1 Worst-case tsu	N/A	None	4.191 ns	A[15]	Areg[15]
2 Worst-case tco	N/A	None	7.181 ns	Zreg[8]	Z[8]
3 Worst-case th	N/A	None	-0.558 ns	Sel	SelR
4 Clock Setup: 'Clock'	-0.469 ns	300.03 MHz (period = 3.333 ns)	263.02 MHz (period = 3.802 ns)	AddSubR	Overflow~reg0
5 Clock Hold: 'Clock'	1.175 ns	300.03 MHz (period = 3.333 ns)	N/A	Areg[0]	Zreg[0]
6 Total number of failed paths					

Figure 9. The timing constraint cannot be met.

The specified f_{max} of 300 MHz cannot be achieved because one or more paths in the circuit have long propagation delays. To locate the most critical path highlight the Clock Setup entry in the table by clicking on it. Then, right-click to get the pop-up menu shown in Figure 10. Select **Locate** > **Locate in RTL Viewer** which will cause the RTL Viewer to display the critical path as presented in Figure 11. Note that this path begins at flip-flop *AddSubR* and ends at the *Overflow* flip-flop.

Type	Slack	Required Time	Actual Time	From	To	Fro Clo
1 Worst-case tsu	N/A	None	4.191 ns	A[15]	Areg[15]	
2 Worst-case tco	N/A	None	7.181 ns	Zreg[8]	Z[8]	Clo
3 Worst-case th	N/A	None	-0.558 ns	Sel	SelR	
4 Clock Setup: 'Clock'	-0.469 ns	300.03 MHz (period = 3.333 ns)	263.02 MHz (period = 3.802 ns)	AddSubR	Overflow~reg0	Clo
5 Clock Hold: 'Clock'	1.175 ns	300.03 MHz (period = 3.333 ns)	N/A	Areg[0]	Zreg[0]	Clo
6 Total number of failed						

Figure 10. Locate the critical path.

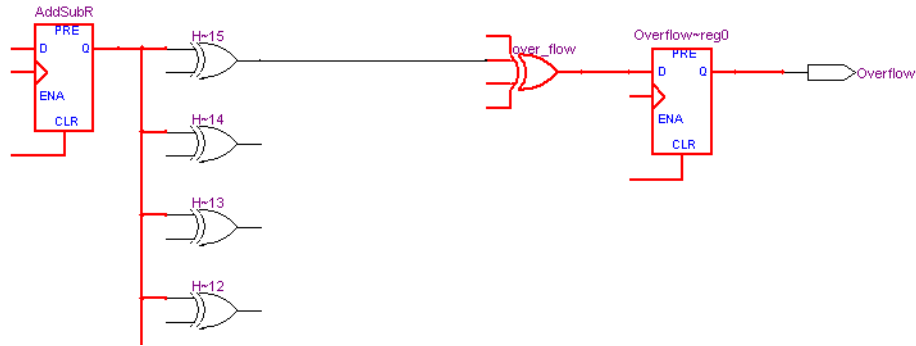


Figure 11. Path for which the timing constraint cannot be met.

It is likely that there are other paths that make it impossible to meet the specified constraint. To identify these paths choose Clock Setup: 'Clock' on the left side of the Compilation Report in Figure 9. As seen in Figure 12, there are 10 paths with propagation delays that are too long. Observe a column labeled Slack. The term *slack* is used to indicate the margin by which a timing requirement is met or not met. In the top row in Figure 12 we see that the timing delays along the path from the *AddSubR* flip-flop to the *Overflow* flip-flop are 0.469 ns longer than the maximum of 4 ns that is the period of the 250-MHz clock specified as the *f_{max}* constraint.

	Slack	Actual fmax (period)	From	To
1	-0.469 ns	253.02 MHz (period = 3.952 ns)	AddSubR	Overflow~reg0
2	-0.411 ns	267.09 MHz (period = 3.744 ns)	Zreg[5]	Overflow~reg0
3	-0.285 ns	276.40 MHz (period = 3.618 ns)	Zreg[0]	Overflow~reg0
4	-0.270 ns	277.55 MHz (period = 3.603 ns)	Self	Overflow~reg0
5	-0.247 ns	279.33 MHz (period = 3.580 ns)	Zreg[1]	Overflow~reg0
6	-0.191 ns	283.77 MHz (period = 3.524 ns)	Zreg[2]	Overflow~reg0
7	-0.152 ns	286.94 MHz (period = 3.485 ns)	Zreg[4]	Overflow~reg0
8	-0.146 ns	287.44 MHz (period = 3.479 ns)	Zreg[3]	Overflow~reg0
9	-0.063 ns	294.46 MHz (period = 3.396 ns)	Zreg[6]	Overflow~reg0
10	-0.049 ns	295.68 MHz (period = 3.382 ns)	Zreg[11]	Overflow~reg0
11	0.019 ns	301.75 MHz (period = 3.314 ns)	Breg[2]	Overflow~reg0
12	0.029 ns	302.66 MHz (period = 3.304 ns)	Zreg[7]	Overflow~reg0
13	0.118 ns	311.04 MHz (period = 3.215 ns)	Breg[4]	Overflow~reg0
14	0.123 ns	311.53 MHz (period = 3.210 ns)	Breg[0]	Overflow~reg0
15	0.163 ns	315.46 MHz (period = 3.170 ns)	Areg[0]	Overflow~reg0
16	0.174 ns	316.56 MHz (period = 3.159 ns)	Areg[11]	Overflow~reg0

Figure 12. The longest delay paths.

We have shown how to set the *f_{max}* constraint. The other constraints depicted in the window in Figure 7 can be set in the same way.

4 Timing Simulation

Timing simulation provides a graphical indication of the delays in the implemented circuit, as can be observed from the displayed waveforms. For a discussion of simulation see the tutorial *Quartus II Simulation with Verilog Designs*, which uses the same *addersubtractor* circuit as an example.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Quartus II Simulation with Verilog Designs

This tutorial introduces the basic features of the Quartus[®] II Simulator. It shows how the Simulator can be used to assess the correctness and performance of a designed circuit.

Contents:

- Example Circuit
- Using the Waveform Editor
- Functional Simulation
- Timing Simulation

Quartus[®] II software includes a simulator which can be used to simulate the behavior and performance of circuits designed for implementation in Altera's programmable logic devices. The simulator allows the user to apply test vectors as inputs to the designed circuit and to observe the outputs generated in response. In addition to being able to observe the simulated values on the I/O pins of the circuit, it is also possible to probe the internal nodes in the circuit. The simulator makes use of the Waveform Editor, which makes it easy to represent the desired signals as waveforms.

Doing this tutorial, the reader will learn about:

- Test vectors needed to test the designed circuit
- Using the Quartus II Waveform Editor to draw the test vectors
- Functional simulation, which is used to verify the functional correctness of a synthesized circuit
- Timing simulation, which takes into account propagation delays due to logic elements and interconnecting wiring

This tutorial is aimed at the reader who wishes to simulate circuits defined by using the Verilog hardware description language. An equivalent tutorial is available for the user who prefers the VHDL language.

PREREQUISITES

The reader is expected to have access to a computer that has Quartus II software installed. The detailed examples in the tutorial were obtained using the Quartus II version 5.0, but other versions of the software can also be used.

1 Example Circuit

As an example, we will use the adder/subtractor circuit shown in Figure 1. The circuit can add, subtract, and accumulate n -bit numbers using the 2's complement number representation. The two primary inputs are numbers $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$, and the primary output is $Z = z_{n-1}z_{n-2} \cdots z_0$. Another input is the *AddSub* control signal which causes $Z = A + B$ to be performed when *AddSub* = 0 and $Z = A - B$ when *AddSub* = 1. A second control input, *Sel*, is used to select the accumulator mode of operation. If *Sel* = 0, the operation $Z = A \pm B$ is performed, but if *Sel* = 1, then B is added to or subtracted from the current value of Z . If the addition or subtraction operations result in arithmetic overflow, an output signal, *Overflow*, is asserted.

To make it easier to deal with asynchronous input signals, they are loaded into flip-flops on a positive edge of the clock. Thus, inputs A and B will be loaded into registers *Areg* and *Breg*, while *Sel* and *AddSub* will be loaded into flip-flops *SelR* and *AddSubR*, respectively. The adder/subtractor circuit places the result into register *Zreg*.

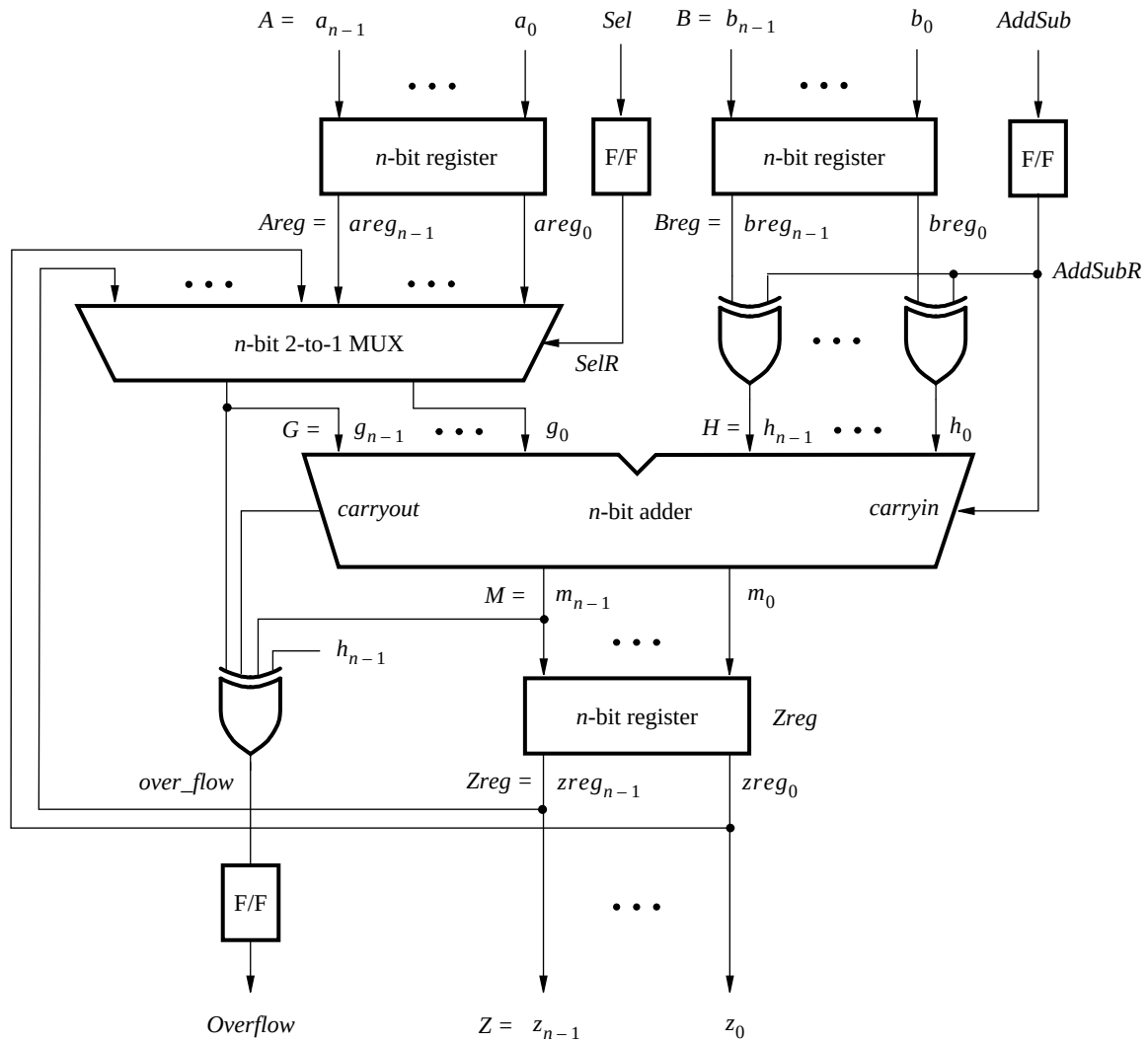


Figure 1. The adder/subtractor circuit.

The required circuit is described by the Verilog code in Figure 2. For our example, we use a 16-bit circuit as specified by $n = 16$. Implement this circuit as follows:

- Create a project *addersubtractor*.
- Include a file *addersubtractor.v*, which corresponds to Figure 2, in the project. For convenience, this file is provided in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages.
- Choose the Cyclone II EP2C35F672C6 device, which is the FPGA chip on Altera's DE2 board.
- Compile the design.

```

// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, H, M, Z;
    wire carryout, over_flow;

// Define combinational logic circuit
    assign H = Breg ^ {n{AddSubR}};
    mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
    adderk nbit_adder (AddSubR, G, H, M, carryout);
    defparam nbit_adder.k = n;
    assign over_flow = carryout ^ G[n-1] ^ H[n-1] ^ M[n-1];
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
            begin
                Areg <= 0; Breg <= 0; Zreg <= 0;
                SelR <= 0; AddSubR <= 0; Overflow <= 0;
            end
        else
            begin
                Areg <= A; Breg <= B; Zreg <= M;
                SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
            end
    endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Selm, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Selm;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Selm)
        if (Selm == 0) F = V;
        else F = W;
endmodule

... continued in Part b

```

Figure 2. Verilog code for the circuit in Figure 1 (Part a).

```
// k-bit adder
module adderk (carryin, X, Y, S, carryout);
    parameter k = 8;
    input [k-1:0] X, Y;
    input carryin;
    output [k-1:0] S;
    output carryout;
    reg [k-1:0] S;
    reg carryout;

    always @(X or Y or carryin)
        {carryout, S} = X + Y + carryin;
endmodule
```

Figure 2. Verilog code for the circuit in Figure 1 (Part b).

2 Using the Waveform Editor

Quartus II software includes a simulation tool that can be used to simulate the behavior of a designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. It is also necessary to specify the outputs, as well as possible internal points in the circuit, which the designer wishes to observe. The simulator applies the test vectors to the model of the implemented circuit and determines the expected response. We will use the Quartus II Waveform Editor to draw the test vectors, as follows:

1. Open the Waveform Editor window by selecting **File > New**, which gives the window shown in Figure 3. Click on the **Other Files** tab to reach the window displayed in Figure 4. Choose **Vector Waveform File** and click OK.

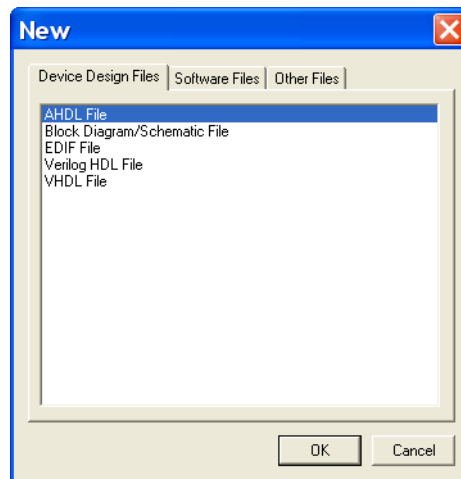


Figure 3. Need to prepare a new file.

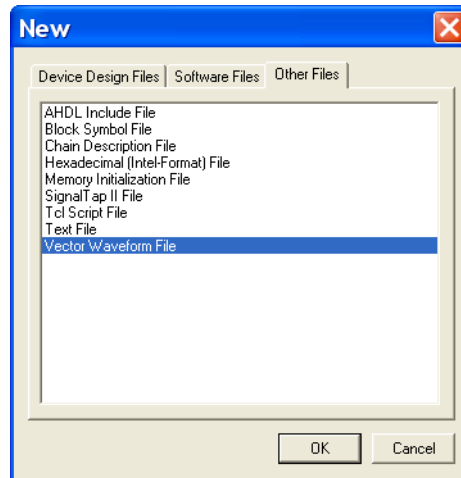


Figure 4. Choose to prepare a test-vector file.

2. The Waveform Editor window is depicted in Figure 5. Save the file under the name *addersubtractor.vwf*; note that this changes the name in the displayed window. In this figure, we have set the desired simulation to run from 0 to 180 ns by selecting **Edit > End Time** and entering 180 ns in the dialog box that pops up. Selecting **View > Fit in Window** displays the entire simulation range of 0 to 180 ns in the window, as shown. Resize the window to its maximum size.

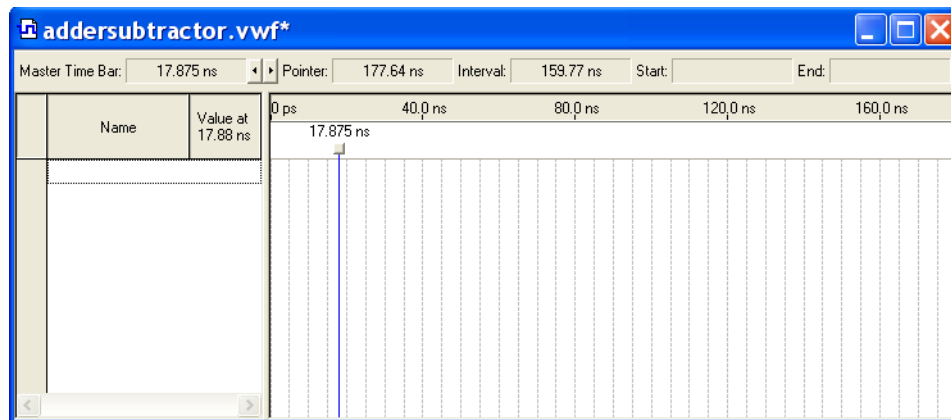


Figure 5. The Waveform Editor window.

3. Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert Node or Bus** to open the window in Figure 6. It is possible to type the full hierarchical name of a signal (pin) into the Name box, but it is easier to click on the button labeled **Node Finder** to open the window in Figure 7. The Node Finder utility has a filter used to indicate what type nodes are to be found. Since we are interested in input and output pins, set the filter to **Pins: all**. Click the **List** button to find the pin names as indicated on the left side of the figure. Observe that the input and output signals *A*, *B*, and *Z* can be selected either as individual nodes (denoted by bracketed subscripts) or as 16-bit vectors, which is a more convenient form.

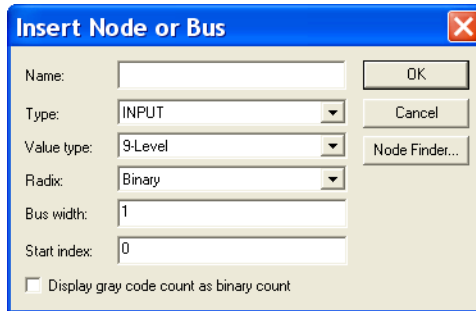


Figure 6. The Insert Node or Bus dialogue.

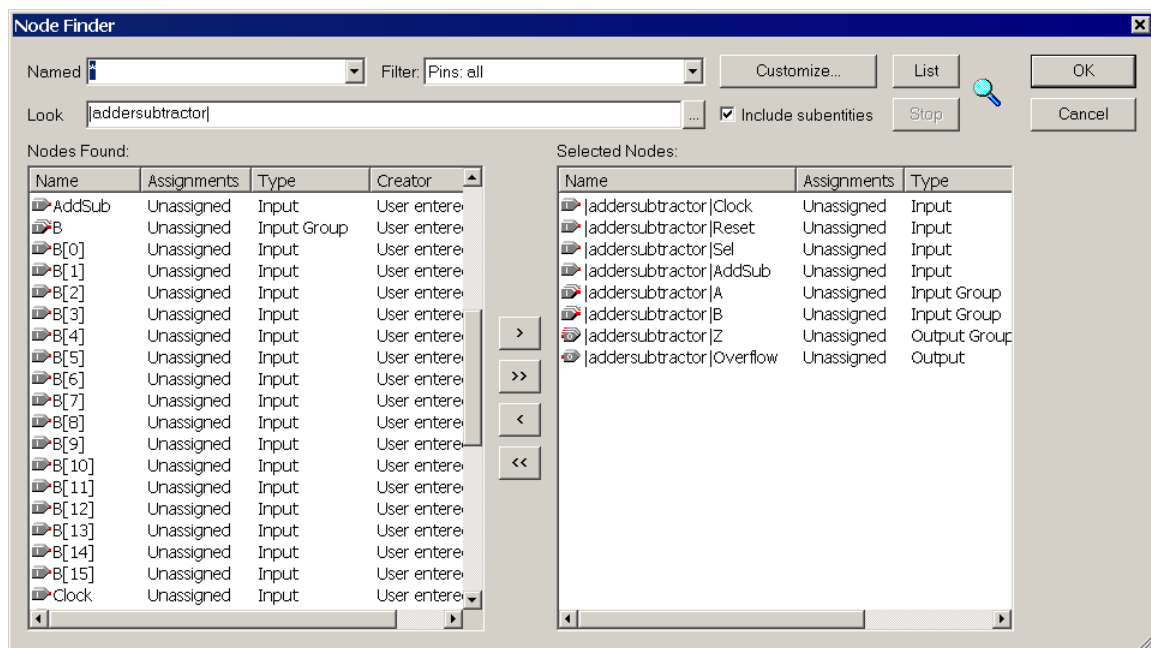


Figure 7. Selecting nodes to insert into the Waveform Editor.

Use the scroll bar inside the Nodes Found box in Figure 7 to find the *Clock* signal. Click on this signal and then click the > sign in the middle of the window to add it to the Selected Nodes box on the right side of the figure. Do the same for *Reset*, *Sel*, and *AddSub*. Then choose vectors *A*, *B* and *Z*, as well as the output *Overflow*, in the same way. (Several nodes can be selected simultaneously in a standard Windows manner.) Click OK to close the Node Finder window, and then click OK in the window of Figure 6. This leaves a fully displayed Waveform Editor window, as shown in Figure 8. If you did not select the nodes in the same order as displayed in Figure 8, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click on the node name (in the Name column) and release the mouse button. The waveform is now highlighted to show the selection. Click again on the waveform and drag it up or down in the Waveform Editor.

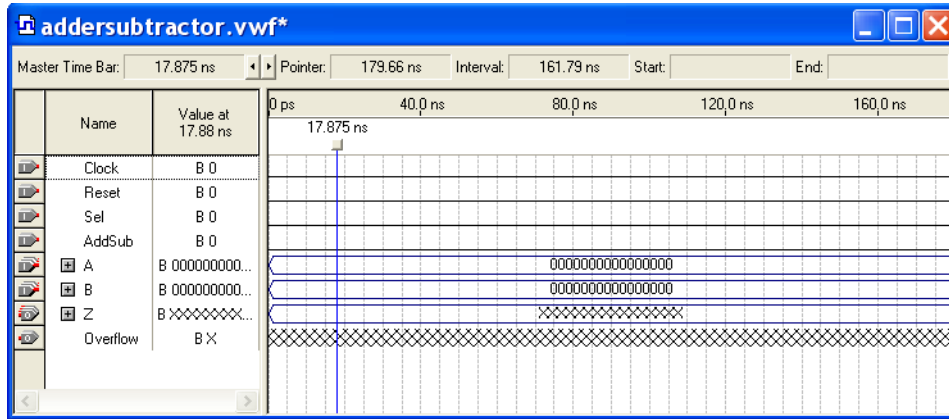


Figure 8. The nodes needed for simulation.

- We will now specify the logic values to be used for the input signals during simulation. The logic values at the outputs *Z* and *Overflow* will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing View > Snap to Grid). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This *reference line* is used in analyzing the timing of a circuit, as described later; move it to the *time* = 0 position. The waveforms can be drawn using the Selection Tool, which is activated by selecting the icon  in the toolbar, or the Waveform Editing Tool, which is activated by the icon . In the instructions below, we will use the Selection Tool.

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. The number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations. We will choose a very small set of input test vectors, which is not sufficient to simulate the circuit properly but is adequate for tutorial purposes. We will use eight 20-ns time intervals to apply the test vectors as shown in Figure 9. The values of signals *Reset*, *Sel*, *AddSub*, *A* and *B* are applied at the input pins as indicated in the figure. The value of *Z* at time t_i is a function of the inputs at time t_{i-1} . When *Sel* = 1, the accumulator feedback loop is activated so that the current value of *Z* (rather than *A*) is used to compute the new value of *Z*.

Time	<i>Reset</i>	<i>Sel</i>	<i>AddSub</i>	<i>A</i>	<i>B</i>	<i>Z</i>
t_0	1	0	0	0	0	0
t_1	0	0	0	54	1850	0
t_2	0	0	1	132	63	1904
t_3	0	0	0	0	0	69
t_4	0	0	1	750	120	0
t_5	0	1	0	0	7000	630
t_6	0	1	0	0	30000	7630
t_7	0	1	0	0	0	37630

Figure 9. The required testing behavior.

The effect of the test vectors in Figure 9 is to perform the following computation:

$$\begin{aligned}
 t_0 &: \text{Reset} \\
 t_1 &: Z(t_1) = 0 \\
 t_2 &: Z(t_2) = A(t_1) + B(t_1) = 54 + 1850 = 1904 \\
 t_3 &: Z(t_3) = A(t_2) - B(t_2) = 132 - 63 = 69 \\
 t_4 &: Z(t_4) = A(t_3) + B(t_3) = 0 + 0 = 0 \\
 t_5 &: Z(t_5) = A(t_4) - B(t_4) = 750 - 120 = 630 \\
 t_6 &: Z(t_6) = Z(t_5) + B(t_5) = 630 + 7000 = 7630 \\
 t_7 &: Z(t_7) = Z(t_6) + B(t_6) = 7630 + 30000 = 37630 \text{ (overflow)}
 \end{aligned}$$

Initially, the circuit is reset asynchronously. Then for two clock cycles the output Z is first the sum and then the difference of the values of A and B at that time. This is followed by setting both A and B to zero to clear the contents of register Z . Then, the accumulator feedback path is tested in the next three clock cycles by performing the computation

$$Z = A(t_4) - B(t_4) + B(t_5) + B(t_6)$$

using the values of A and B shown above.

We can generate the desired input waveforms as follows. Click on the waveform name for the *Clock* node. Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for defining the clock, or setting the selected signal to 0, 1, unknown (X), high impedance (Z), don't care (DC), and inverting its existing value (INV). Each command can be activated by using the **Edit > Value** command, or via the toolbar for the Waveform Editor. The Edit menu can also be opened by right-clicking on a waveform name.

With the *Clock* signal highlighted, click on the **Overwrite Clock** icon  in the toolbar. This leads to the pop-up window in Figure 10. Enter the clock period value of 20 ns, make sure that the phase is 0 and the duty cycle is 50 percent, and click OK. The desired clock signal is now displayed in the Waveform window.

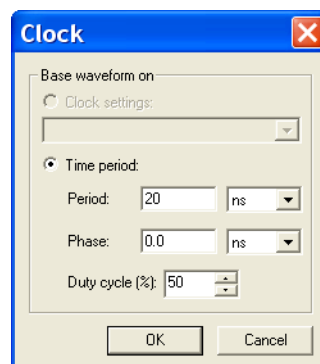


Figure 10. Definition of the clock period, phase and duty cycle.

We will assume, for simplicity of timing, that the input signals change coincident with the negative edges of the clock. To reset the circuit, set *Reset* = 1 in the time interval 0 to 20 ns. Do this by pressing the mouse at the start of the interval and dragging it to its end, which highlights the selected interval, and choosing the logic value 1 in the toolbar. Make *Sel* = 1 from 100 to 160 ns, and *AddSub* = 1 in periods 40 to 60 ns and 80 to 100 ns. This should produce the image in Figure 11.

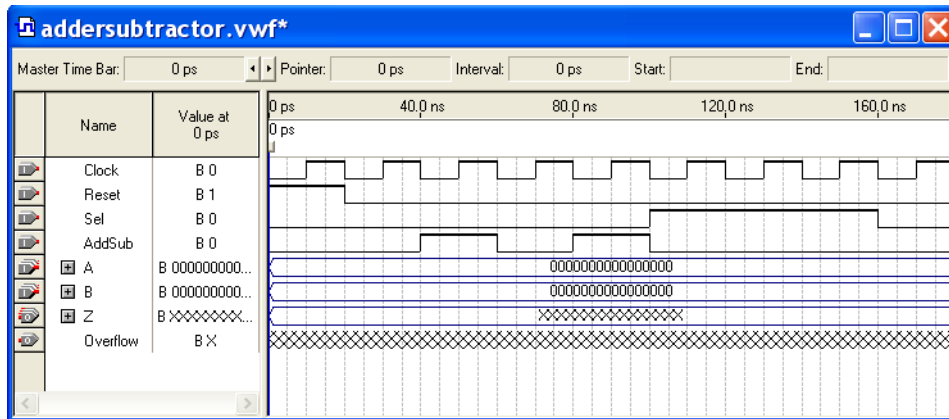


Figure 11. Setting of test values for the control signals.

5. Vectors can be treated as either octal, hexadecimal, signed decimal, or unsigned decimal numbers. The vectors *A*, *B*, and *Z* are initially treated as binary numbers. For our purpose it is convenient to treat them as signed decimal numbers, so right-click on *A* and select **Properties** in the pop-up box to get to the window displayed in Figure 12. Choose signed decimal as the radix, make sure that the bus width is 16 bits, and click OK. In the same manner, declare that *B* and *Z* should be treated as signed decimal numbers.

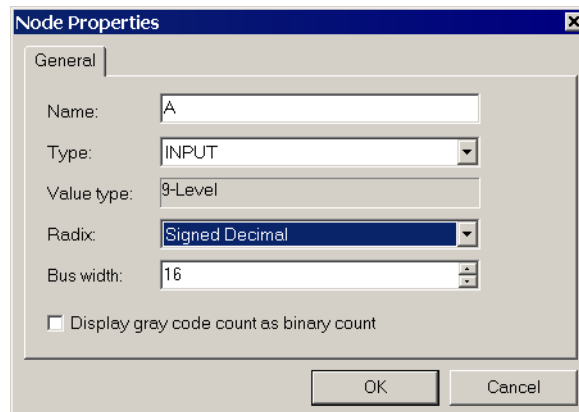


Figure 12. Definition of node properties.

The default value of *A* is 0. To assign specific values in various intervals proceed as follows. Select (highlight) the interval from 20 to 40 ns and press the **Arbitrary Value** icon  in the toolbar, to bring up the pop-up window in Figure 13. Enter the value 54 and click OK. Similarly, for the subsequent 20-ns intervals set *A* to the values 132, 0, 750, and then 0 to the end. Set the corresponding values of *B* to 1850, 63, 0, 120, 7000, 30000, and 0, to generate the waveforms depicted in Figure 14. Observe that the outputs *Z* and *Overflow* are displayed as having unknown values at this time, which is indicated by a hashed pattern; their values will be determined during simulation. Save the file.

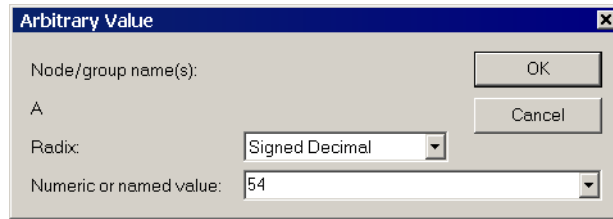


Figure 13. Specifying a value for a multibit signal.

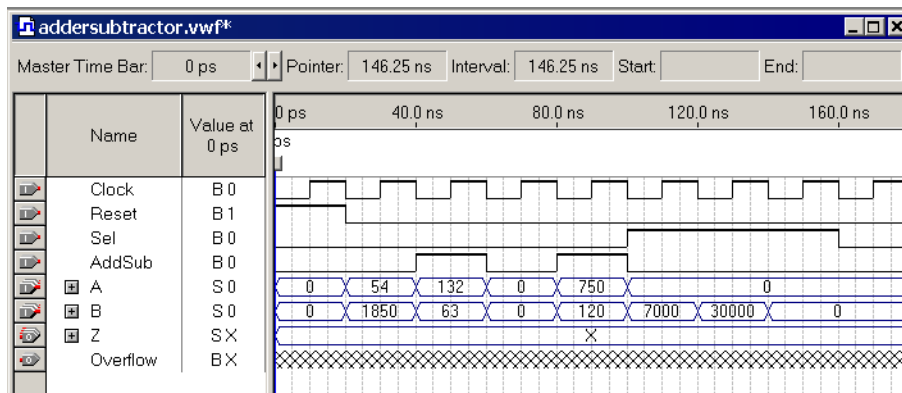


Figure 14. The specified input test vectors.

Another convenient mechanism for changing the input waveforms is provided by the Waveform Editing tool, which is activated by the icon . When the mouse is dragged over some time interval in which the waveform is 0 (1), the waveform will be changed to 1 (0). Experiment with this feature on signal *AddSub*.

3 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and inter-connection wires are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit.

3.1 Functional Simulation

To perform the functional simulation, select **Assignments > Settings** to open the Settings window shown in Figure 15. On the left side of this window click on **Simulator** to display the window in Figure 16, choose **Functional** as the simulation mode, and click **OK**. The Quartus II simulator takes the inputs and generates the outputs defined in the *addersubtractor.vwf* file. Before running the functional simulation it is necessary to create the required netlist, which is done by selecting **Processing > Generate Functional Simulation Netlist**.

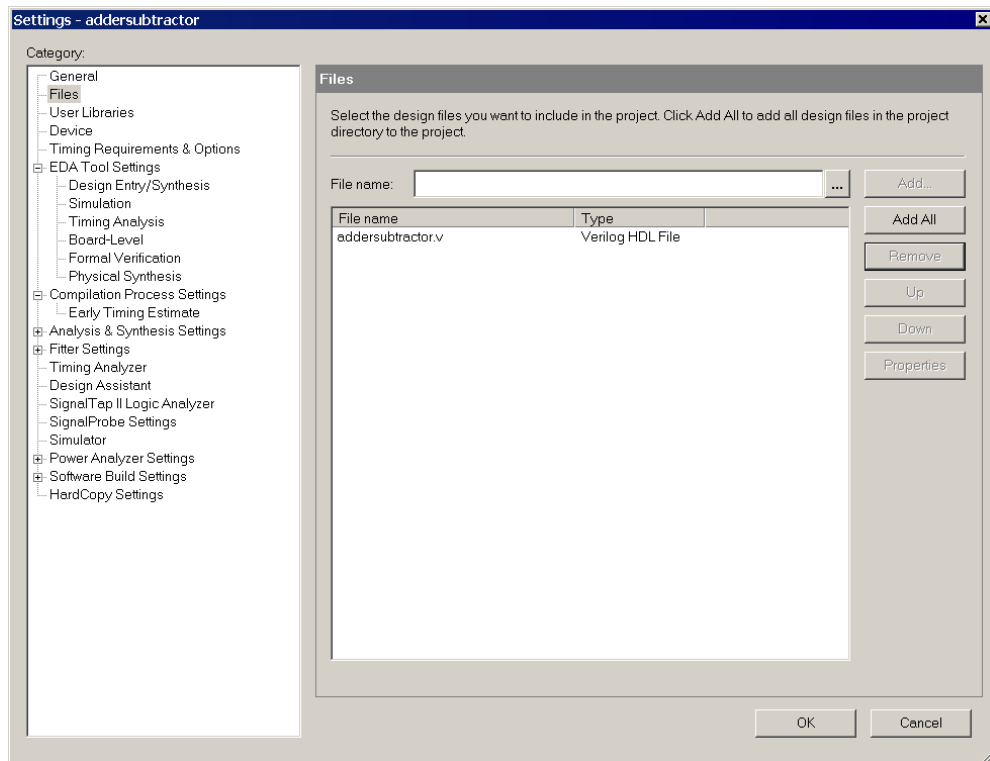


Figure 15. Settings window.

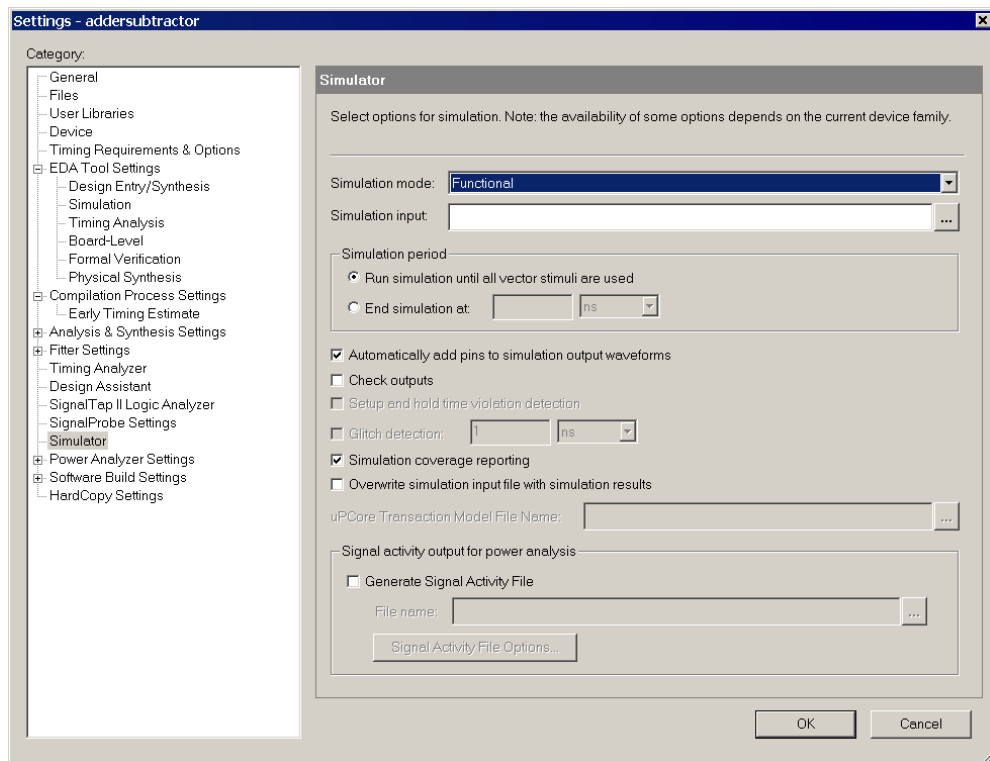


Figure 16. Specifying the simulation mode.

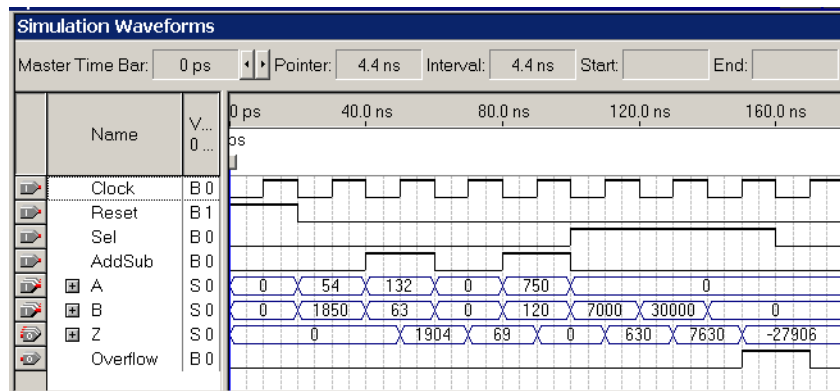


Figure 17. The result of functional simulation.

A simulation run is started by **Processing > Start Simulation**, or by using the icon . At the end of the simulation, Quartus II software indicates its successful completion and displays a Simulation Report illustrated in Figure 17. As seen in the figure, the Simulator creates waveforms for the outputs *Z* and *Overflow*. As expected, the values of *Z* indicate the correct sum or difference of the applied inputs one clock cycle later because of the registers in the circuit. Note that the last value of *Z* is incorrect because the expected sum of 37630 is too big to be represented as a signed number in 16 bits, which is indicated by the *Overflow* signal being set to 1.

In this simulation, we considered only the input and output signals, which appear on the pins of the FPGA chip. It is also possible to look at the behavior of internal signals. For example, let us consider the registered signals *SelR*, *AddSubR*, *Areg*, *Breg*, and *Zreg*. Open the *addersubtractor.vwf* file and activate the Node Finder window, as done for Figure 6. The filter in Figure 6 specified **Pins: all**. There are several other choices. To find the registered signals, set the filter to **Registers: post-fitting** and press **List**. Figure 18 shows the result. Select the signals *SelR*, *AddSubR*, *Areg*, *Breg*, and *Zreg* for inclusion in the *addersubtractor.vwf* file, and specify that *Areg*, *Breg*, and *Zreg* have to be displayed as signed decimal numbers, thus obtaining the display in Figure 19. Save the file and simulate the circuit using these waveforms, which should produce the result shown in Figure 20.

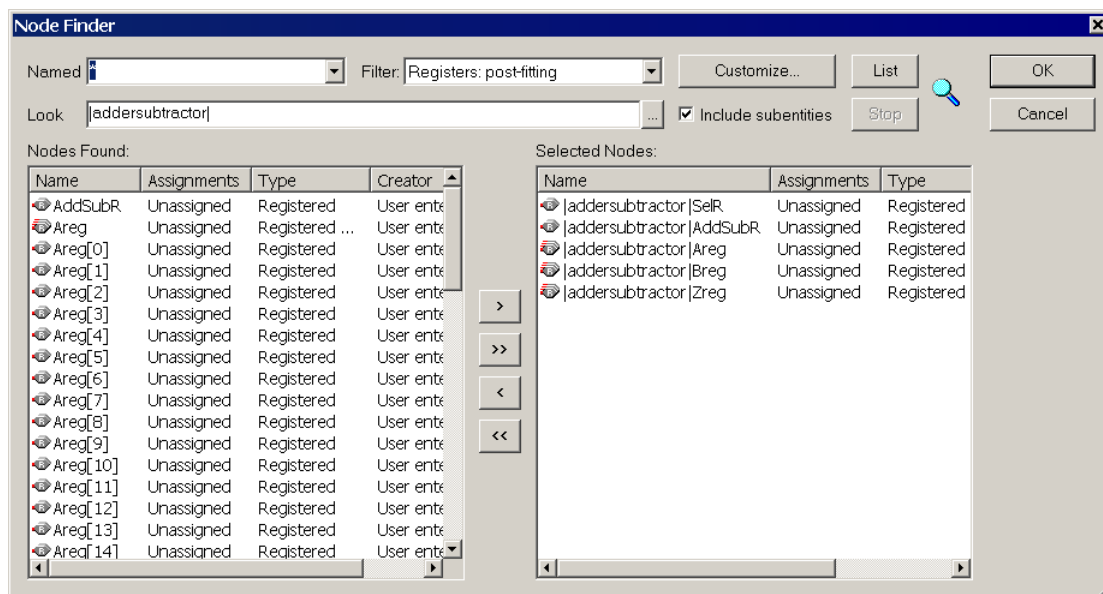


Figure 18. Finding the registered signals.

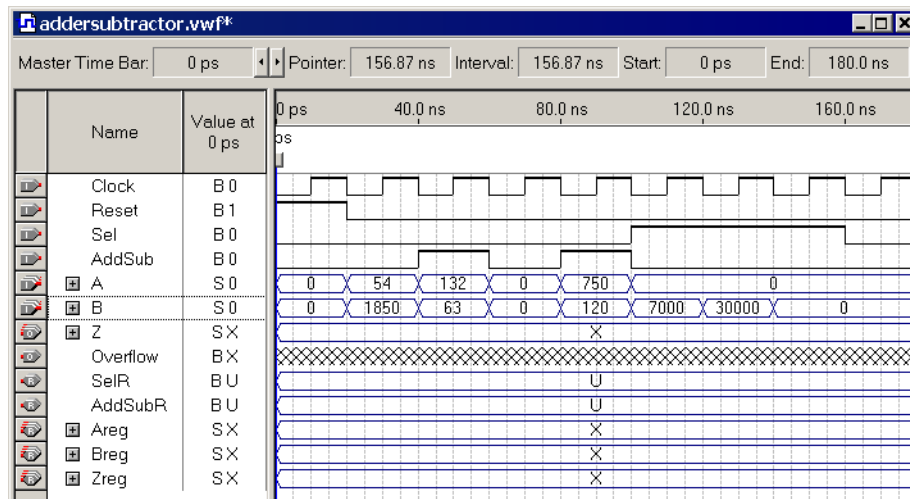


Figure 19. Inclusion of registered signals in the test.

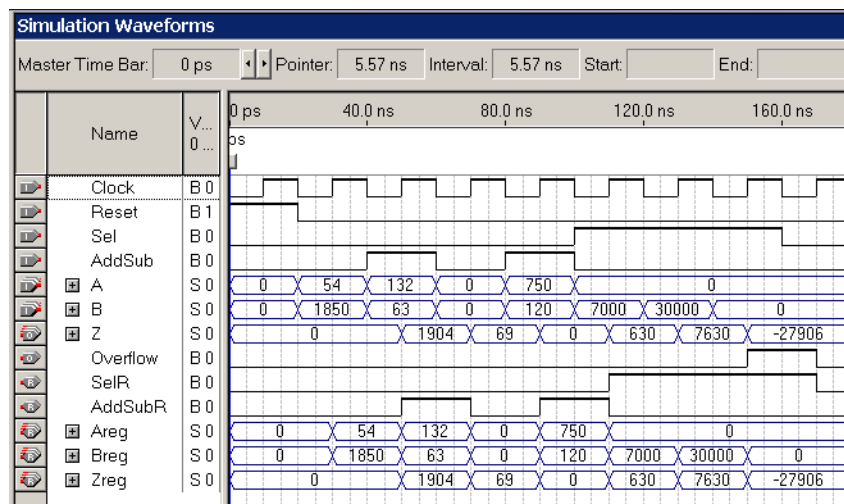


Figure 20. The result of new simulation.

3.2 Timing Simulation

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how well it performs in terms of speed. Select Assignments > Settings > Simulator to get to the window in Figure 16, choose Timing as the simulation mode, and click OK. Run the simulator, which should produce the waveforms in Figure 21. Observe that there are delays in loading the various registers as well as longer delays in producing valid signals on the output pins.

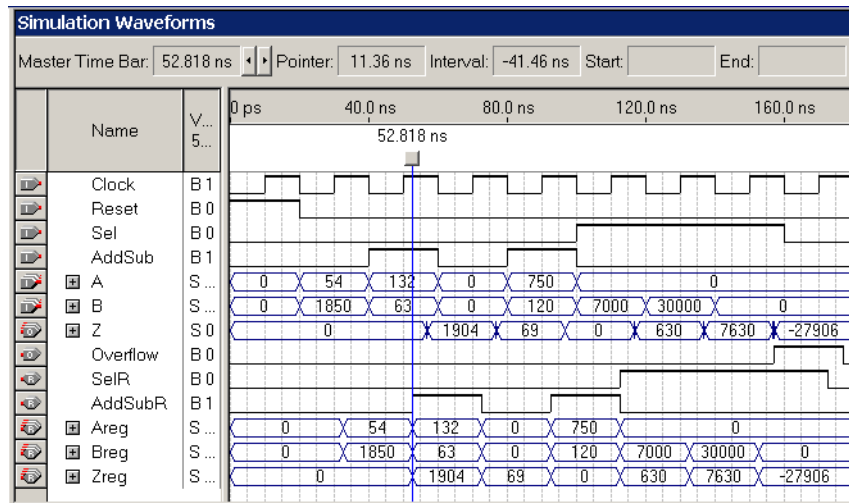


Figure 21. The result of timing simulation.

As an aid in seeing the actual values of the delays, we can use the reference line. Point to the small square handle at the top of the reference line and drag it to the rising edge of the first *AddSubR* pulse, at which time the registers are also loaded, as indicated in the figure. (To make it possible to move the reference line to any point in the waveform display, you may have to turn off the feature View > Snap on Grid.) This operation places the reference line at about the 52.8 ns point, which indicates that it takes 2.8 ns to load the registers after the rising edge of the clock (which occurs at 50 ns). The output *Z* attains its correct value some time after this value has been loaded into *Zreg*. To determine the propagation delay to the output pins, drag the reference line to the point where *Z* becomes valid. This can be done more accurately by enlarging the displayed simulation waveforms by using the Zoom Tool. Left-click on the display to enlarge it and right-click to reduce it. Enlarge the display so that it looks like the image in Figure 22. (After enlarging the image, click on the Selection Tool icon . Position the reference line where *Z* changes to 1904, which occurs at about 57.2 ns. The display indicates that the propagation delay from register *Zreg* to the output pins *Z* is $57.2 - 52.8 = 4.4$ ns. It is useful to note that even before we performed this simulation, the Quartus II timing analyzer evaluated various delays in the implemented circuit and reported them in the Compilation Report. From the Compilation Report we can see that the worst case *tco* (Clock to Output Delay) for the *Z* output (pin *z3*) was estimated as 7.18 ns; this delay can be found by zooming into the simulation results at the point where *Z* changes to the value 7630.

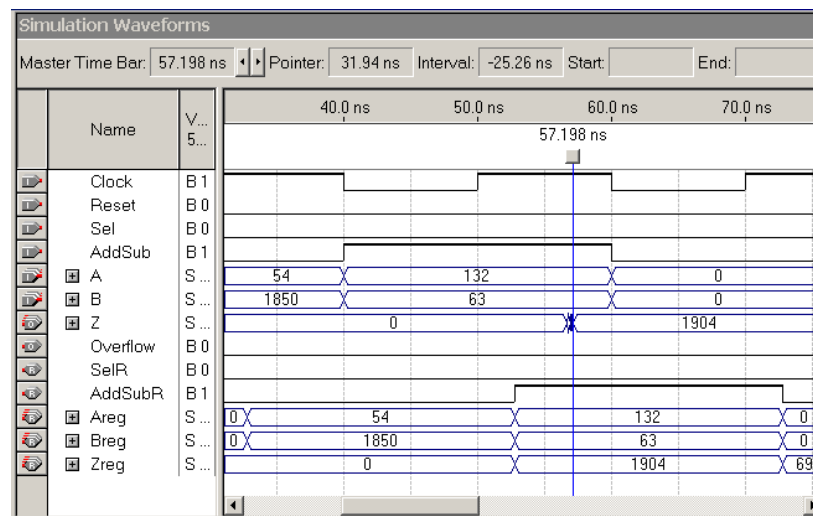


Figure 22. An enlarged image of the simulated waveforms.

In this discussion, we have used the numbers obtained during our simulation run. The user is likely to obtain somewhat different numbers, depending on the version of Quartus II software that is used.

Copyright ©2005 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Introduction to the Altera SOPC Builder

Using Verilog Design

This tutorial presents an introduction to Altera's SOPC Builder software, which is used to implement a system that uses the Nios II processor on an Altera FPGA device. The system development flow is illustrated by giving step-by-step instructions for using the SOPC Builder in conjunction with the Quartus[®] II software to implement a simple system.

The last step in the development process involves configuring the designed circuit in an actual FPGA device, and running an application program. To show how this is done, it is assumed that the user has access to the Altera DE2 Development and Education board connected to a computer that has Quartus II and Nios[®] II software installed.

The screen captures in the tutorial were obtained using the Quartus II version 5.1; if other versions of the software are used, some of the images may be slightly different.

Contents:

Nios II System

Altera's SOPC Builder

Integration of the Nios II System into a Quartus II Project

Running the Application Program

Altera's Nios II is a soft processor, defined in a hardware description language, which can be implemented in Altera's FPGA devices by using the Quartus® II CAD system. To implement a useful system it is necessary to add other functional units such as memories, input/output interfaces, timers, and communications interfaces. To facilitate the implementation of such systems, it is useful to have computer-aided-design (CAD) software for implementing a system-on-a-programmable-chip (SOPC). Altera's SOPC Builder is the software needed for this task.

This tutorial provides a basic introduction to Altera's SOPC Builder, which will allow the reader to quickly implement a simple Nios II system on the Altera DE2 board. For a fuller treatment of the SOPC Builder, the reader can consult the *Nios II Hardware Development Tutorial*. A complete description of the SOPC Builder can be found in the *Quartus II Handbook Volume 4: SOPC Builder*. These documents are available on the Altera web site.

1 Nios II System

A Nios II system can be implemented on the DE2 board as shown in Figure 1.

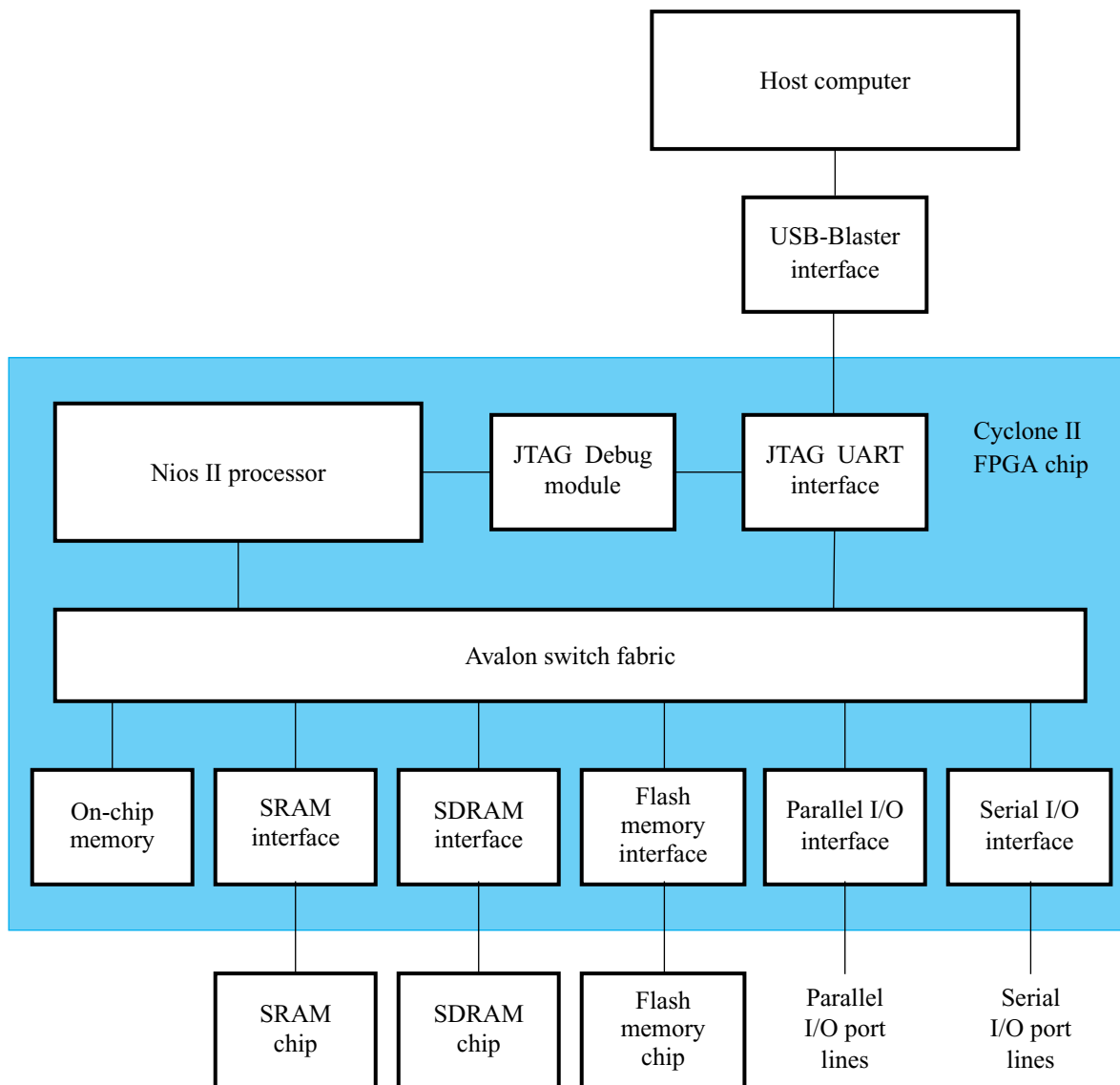


Figure 1. A Nios II system implemented on the DE2 board.

The Nios II processor and the interfaces needed to connect to other chips on the DE2 board are implemented in the Cyclone II FPGA chip. These components are interconnected by means of the interconnection network called the Avalon Switch Fabric. The memory blocks in the Cyclone II device can be used to provide an on-chip memory for the Nios II processor. The SRAM, SDRAM and Flash memory chips on the DE2 board are accessed through the appropriate interfaces. Parallel and serial input/output interfaces provide typical I/O ports used in computer systems. A special JTAG UART interface is used to connect to the circuitry that provides a Universal Serial Bus (USB) link to the host computer to which the DE2 board is connected. This circuitry and the associated software is called the *USB-Blaster*. Another module, called the JTAG Debug module, is provided to allow the host computer to control the Nios II system. It makes it possible to perform operations such as downloading programs into memory, starting and stopping execution, setting breakpoints, and collecting real-time execution trace data.

Since all parts of the Nios II system implemented on the FPGA chip are defined by using a hardware description language, a knowledgeable user could write such code to implement any part of the system. This would be an onerous and time consuming task. Instead, one can use the SOPC Builder to implement a desired system simply by choosing the required components and specifying the parameters needed to make each component fit the overall requirements of the system. In this tutorial, we will illustrate the capability of the SOPC Builder by designing a very simple system. The same approach is used to design large systems.

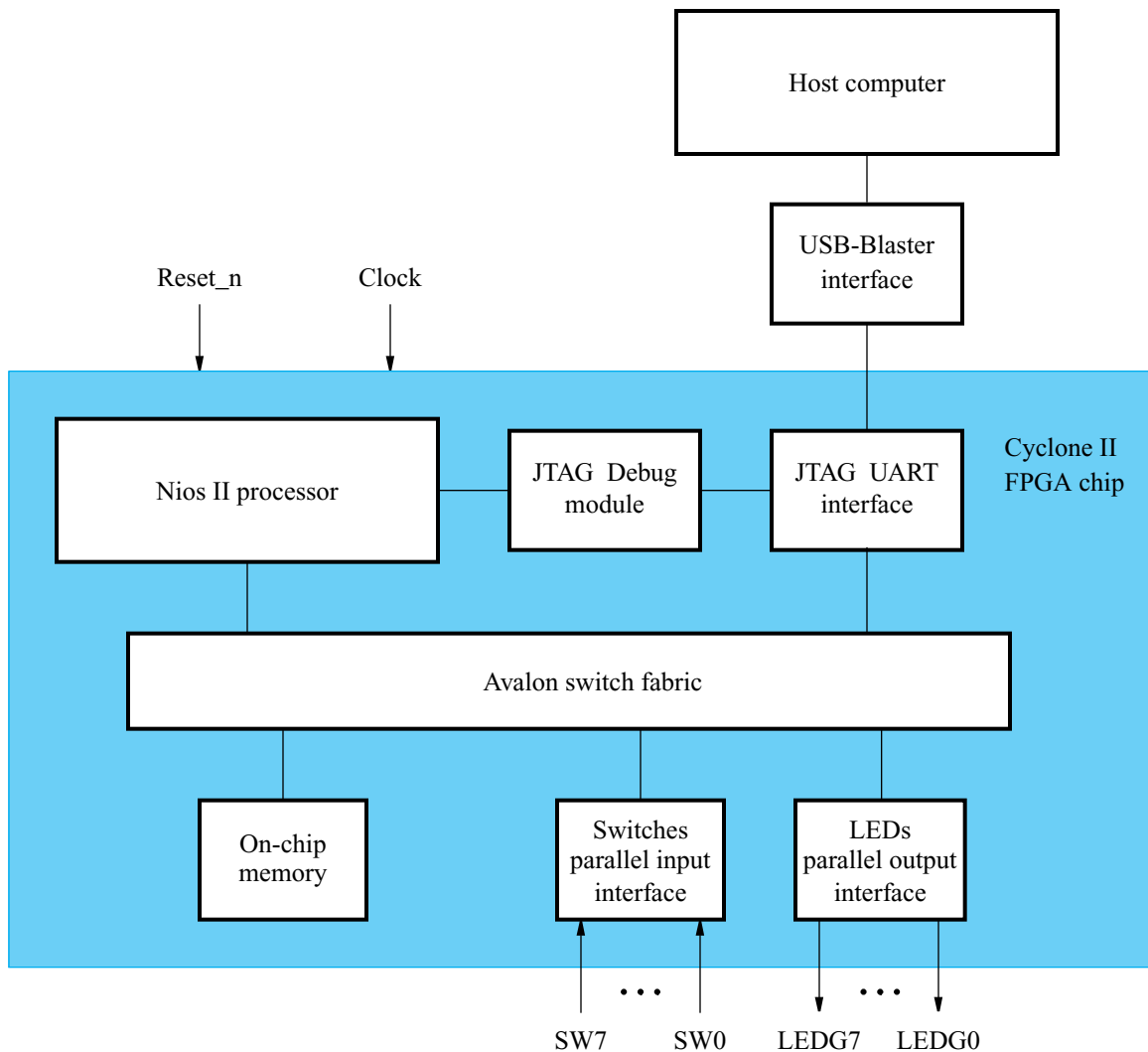


Figure 2. A simple example of a Nios II system.

Our example system is given in Figure 2. The system realizes a trivial task. Eight toggle switches on the DE2 board, $SW7 - 0$, are used to turn on or off the eight green LEDs, $LEDG7 - 0$. The switches are connected to the Nios II system by means of a parallel I/O interface configured to act as an input port. The LEDs are driven by the signals from another parallel I/O interface configured to act as an output port. To achieve the desired operation, the eight-bit pattern corresponding to the state of the switches has to be sent to the output port to activate the LEDs. This will be done by having the Nios II processor execute a program stored in the on-chip memory. Continuous operation is required, such that as the switches are toggled the lights change accordingly.

We will use the SOPC Builder to design the hardware depicted in Figure 2. Next, we will assign the Cyclone II pins to realize the connections between the parallel interfaces and the switches and LEDs which act as I/O devices. Then, we will configure the FPGA to implement the designed system. Finally, we will use the software tool called the *Nios II Debug Client* to assemble, download and execute a Nios II program that performs the desired task.

Doing this tutorial, the reader will learn about:

- Using the SOPC Builder to design a Nios II-based system
- Integrating the designed Nios II system into a Quartus II project
- Implementing the designed system on the DE2 board
- Running an application program on the Nios II processor

2 Altera's SOPC Builder

The SOPC Builder is a tool used in conjunction with the Quartus II CAD software. It allows the user to easily create a system based on the Nios II processor, by simply selecting the desired functional units and specifying their parameters. To implement the system in Figure 2, we have to instantiate the following functional units:

- Nios II processor, which is referred to as a Central Processing Unit (CPU)
- On-chip memory, which consists of the memory blocks in the Cyclone II chip; we will specify a 4-Kbyte memory arranged in 32-bit words
- Two parallel I/O interfaces
- JTAG UART interface for communication with the host computer

To define the desired system, start the Quartus II software and perform the following steps:

1. Create a new Quartus II project for your system. As shown in Figure 3, we stored our project in a directory called *sopc_builder_tutorial*, and we assigned the name *lights* to both the project and its top-level design entity. You can choose a different directory or project name, but be aware that the SOPC Builder software does not permit the use of spaces in file names. For example, an attempt to use a directory name *sopc builder tutorial* would lead to an error. In your project, choose the EP2C35F672C6 chip as the target device, because this is the FPGA on the DE2 board.
2. Select Tools > SOPC Builder, which leads to the pop-up box in Figure 4. Enter *nios_system* as the system name; this will be the name of the system that the SOPC Builder will generate. Choose Verilog as the target HDL, in which the system module will be specified. Click OK to reach the window in Figure 5.

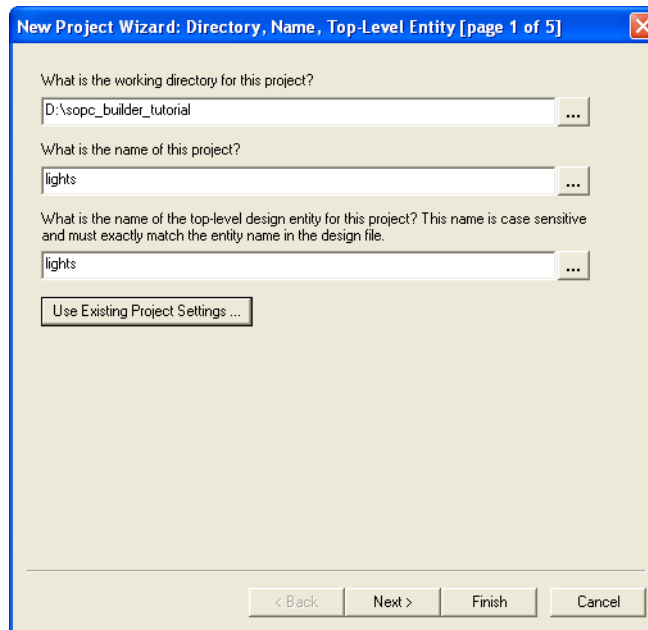


Figure 3. Create a new project.



Figure 4. Create a new Nios II system.

3. Figure 5 displays the System Contents tab of the SOPC Builder, which is used to add components to the system and configure the selected components to meet the design requirements. The available components are listed on the left side of the window. Before choosing our components, examine the area in the figure labeled **Target**. A drop-down list is provided that allows some available Altera boards to be selected. It is not necessary to select a board, and since the DE2 board is not included in the list leave the selection as **Unspecified board**. Next, check the setting for the **Device Family** and ensure that **Cyclone II** is selected.
4. The Nios II processor runs under the control of a clock. For this tutorial we will make use of the 50-MHz clock that is provided on the DE2 board. As shown in Figure 5, it is possible to specify the names and frequency of clock signals in the SOPC Builder display. If not already included in this list, specify a clock named *clk* with the source designated as **External** and the frequency set to 50.0 MHz.

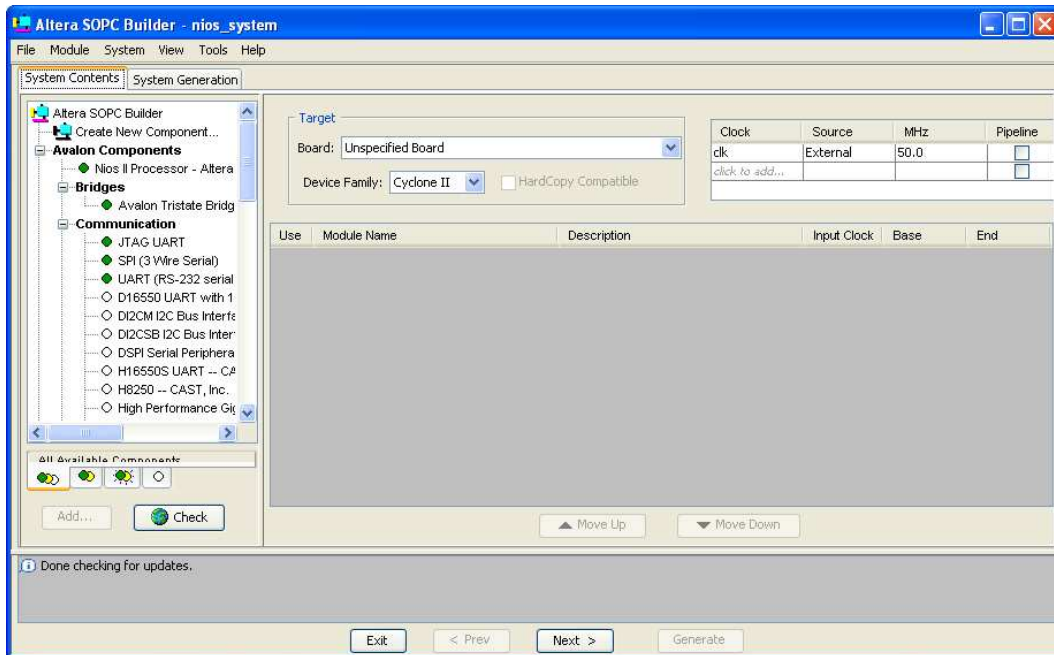


Figure 5. The System Contents tab window.

5. Next, specify the processor as follows:

- On the left side of the window in Figure 5 select Avalon Components > Nios II Processor - Altera Corporation and click Add, which leads to the window in Figure 6.

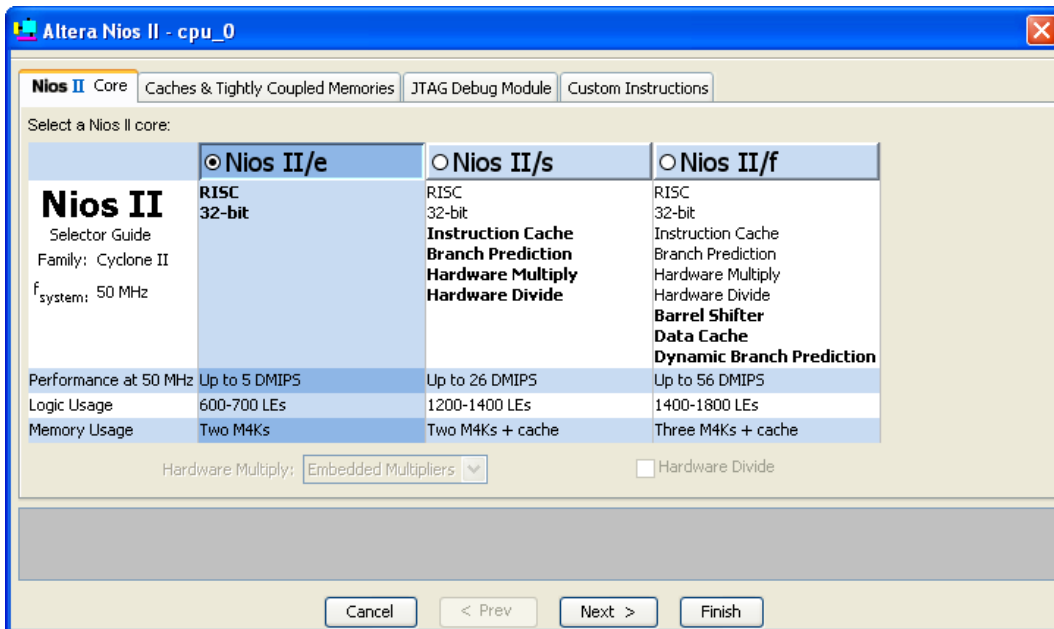


Figure 6. Create a Nios II processor.

- Choose Nios II/e which is the simplest version of the processor. Click **Finish** to return to the window in Figure 5, which now shows the Nios II processor specified as indicated in Figure 7. There may be some warnings or error messages displayed in the SOPC Builder Messages window (at the bottom of the screen), because some parameters have not yet been specified. Ignore these messages as we will provide the necessary data later. Observe also that a new tab called **Nios II More “cpu_0” Settings** appears, which allows further configuration of the processor - we will not use it.

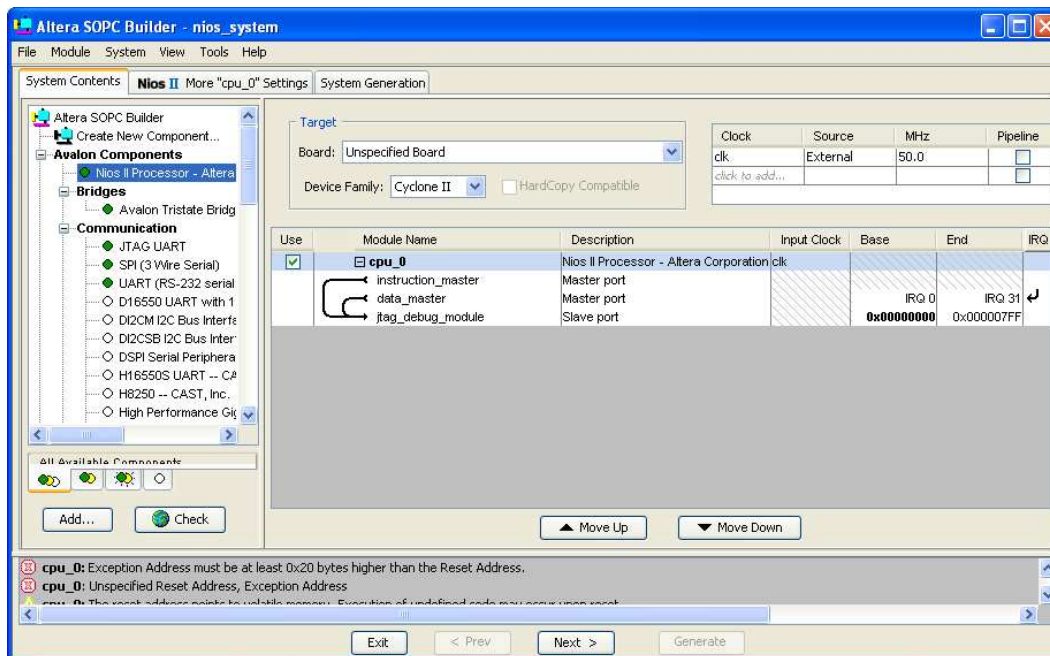


Figure 7. The defined processor.

- To specify the on-chip memory perform the following:
 - Select Avalon Components > Memory > On-Chip Memory (RAM or ROM) and click Add
 - In the On-Chip Memory Configuration Wizard window, shown in Figure 8, set the memory width to 32 bits and the total memory size to 4 Kbytes
 - Do not change the other default settings
 - Click Finish, which returns to the System Contents tab as indicated in Figure 9

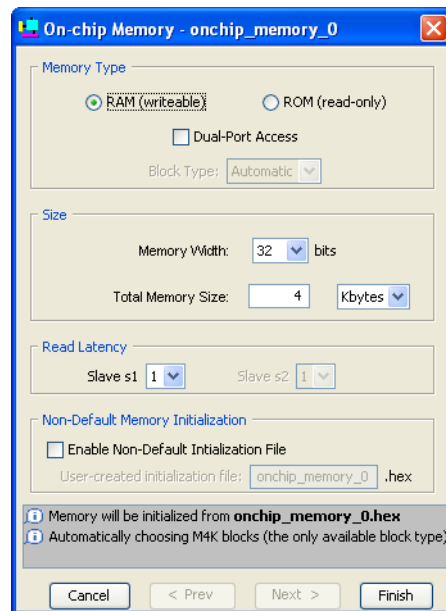


Figure 8. Define the on-chip memory.

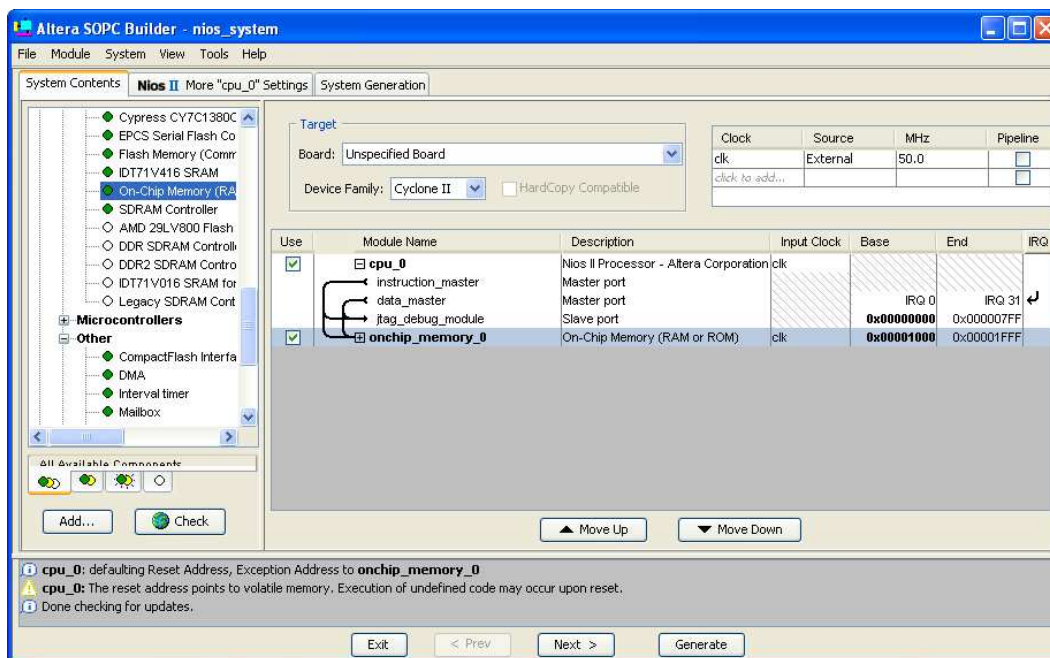


Figure 9. The on-chip memory is included.

7. Specify the input parallel I/O interface as follows:

- Select Avalon Components > Other > PIO (Parallel I/O) and click Add to reach the PIO Configuration Wizard in Figure 10

- Specify the width of the port to be 8 bits and choose the direction of the port to be Input, as shown in the figure
- Click Finish to return to the System Contents tab as given in Figure 11

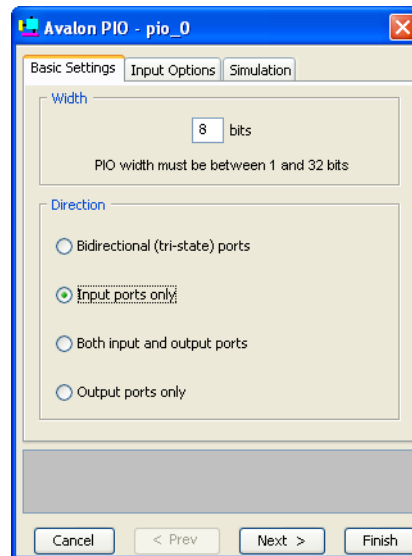


Figure 10. Define a parallel input interface.

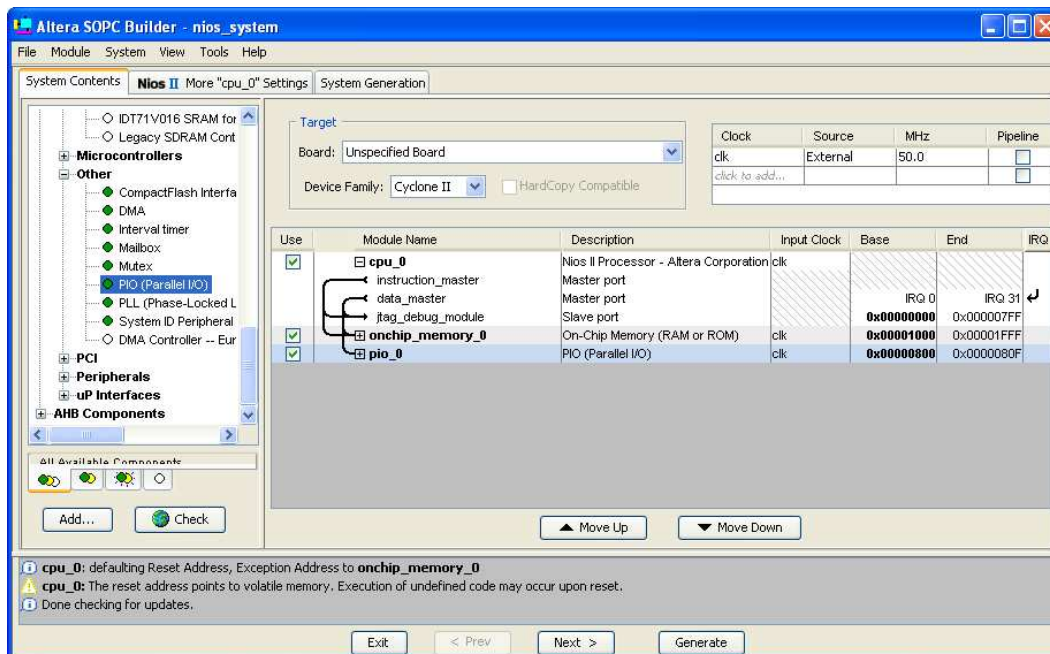


Figure 11. The parallel input interface is included.

8. In the same way, specify the output parallel I/O interface:
 - Select Avalon Components > Other > PIO (Parallel I/O) and click Add to reach the PIO Configuration Wizard again
 - Specify the width of the port to be 8 bits and choose the direction of the port to be Output
 - Click Finish to return to the System Contents tab
9. We wish to connect to a host computer and provide a means for communication between the Nios II system and the host computer. This can be accomplished by instantiating the JTAG UART interface as follows:
 - Select Avalon Components > Communication > JTAG UART and click Add to reach the JTAG UART Configuration Wizard in Figure 12
 - Do not change the default settings
 - Click Finish to return to the System Contents tab

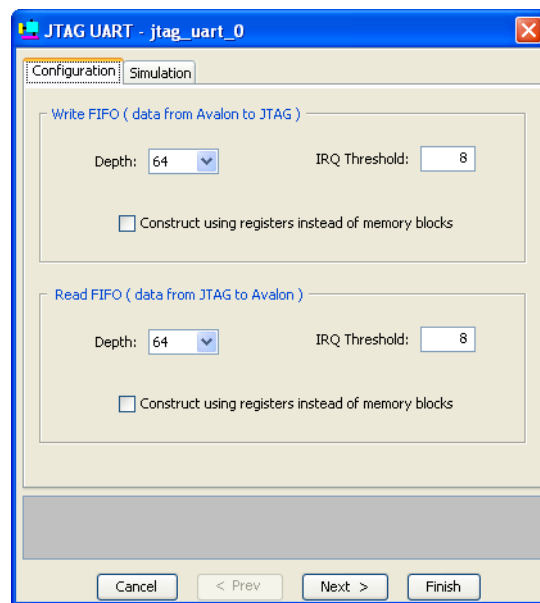


Figure 12. Define the JTAG UART interface.

10. The complete system is depicted in Figure 13. Note that the SOPC Builder automatically chooses names for the various components. The names are not necessarily descriptive enough to be easily associated with the target design, but they can be changed. In Figure 2, we use the names Switches and LEDs for the parallel input and output interfaces, respectively. These names can be used in the implemented system. Right-click on the `pio_0` name and then select **Rename**. Change the name to Switches. Similarly, change `pio_1` to LEDs.
11. The base and end addresses of the various components in the designed system can be assigned by the user, but they can also be assigned automatically by the SOPC Builder. We will choose the latter possibility. So, select the command (using the menus at the top of the SOPC Builder window) **System > Auto-Assign Base Addresses**, which produces the assignment shown in Figure 14.

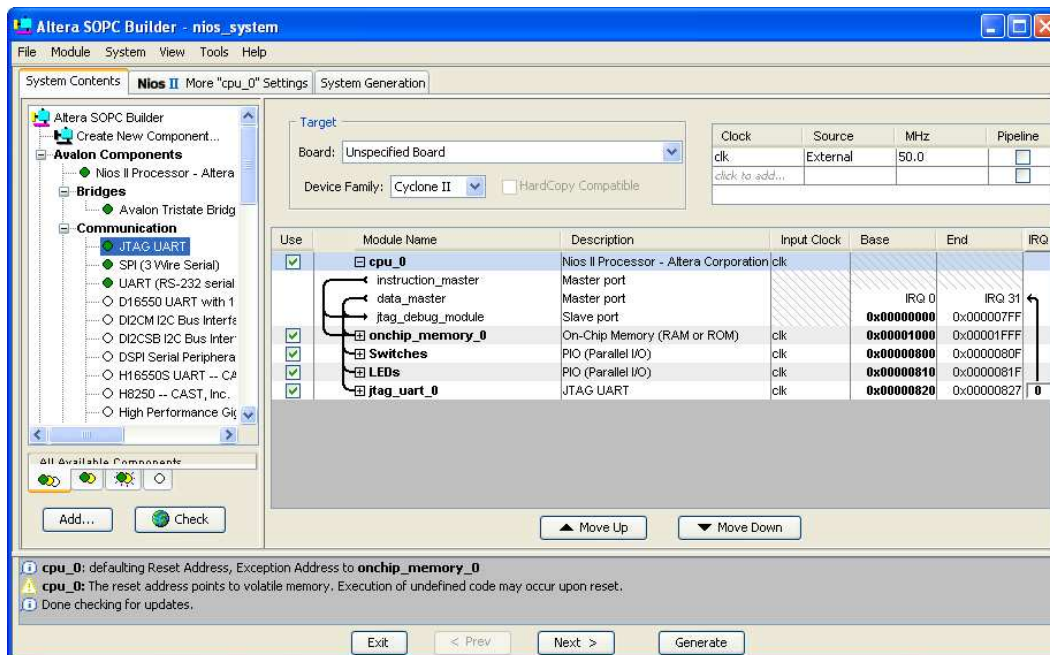


Figure 13. The complete system.

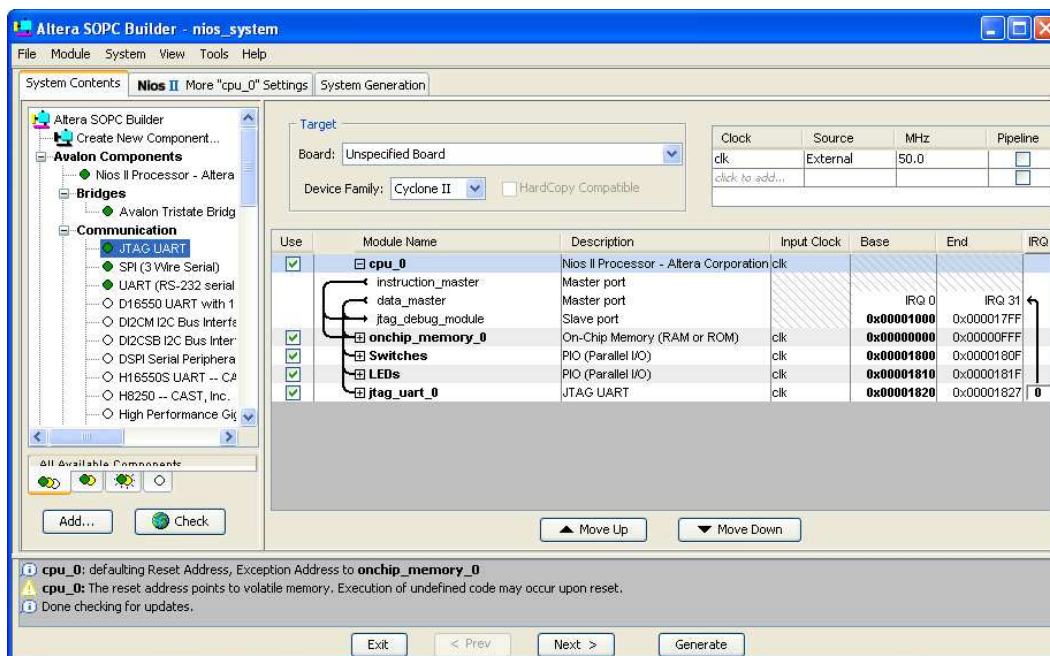


Figure 14. The final specification.

12. Having specified all components needed to implement the desired system, it can now be generated. Select the System Generation tab, which leads to the window in Figure 15. Turn off Simulation - Create simulator project files, because in this tutorial we will not deal with the simulation of hardware. Click

Generate on the bottom of the SOPC Builder window. The generation process produces the messages displayed in the figure. When the message “SUCCESS: SYSTEM GENERATION COMPLETED” appears, click Exit. This returns to the main Quartus II window.

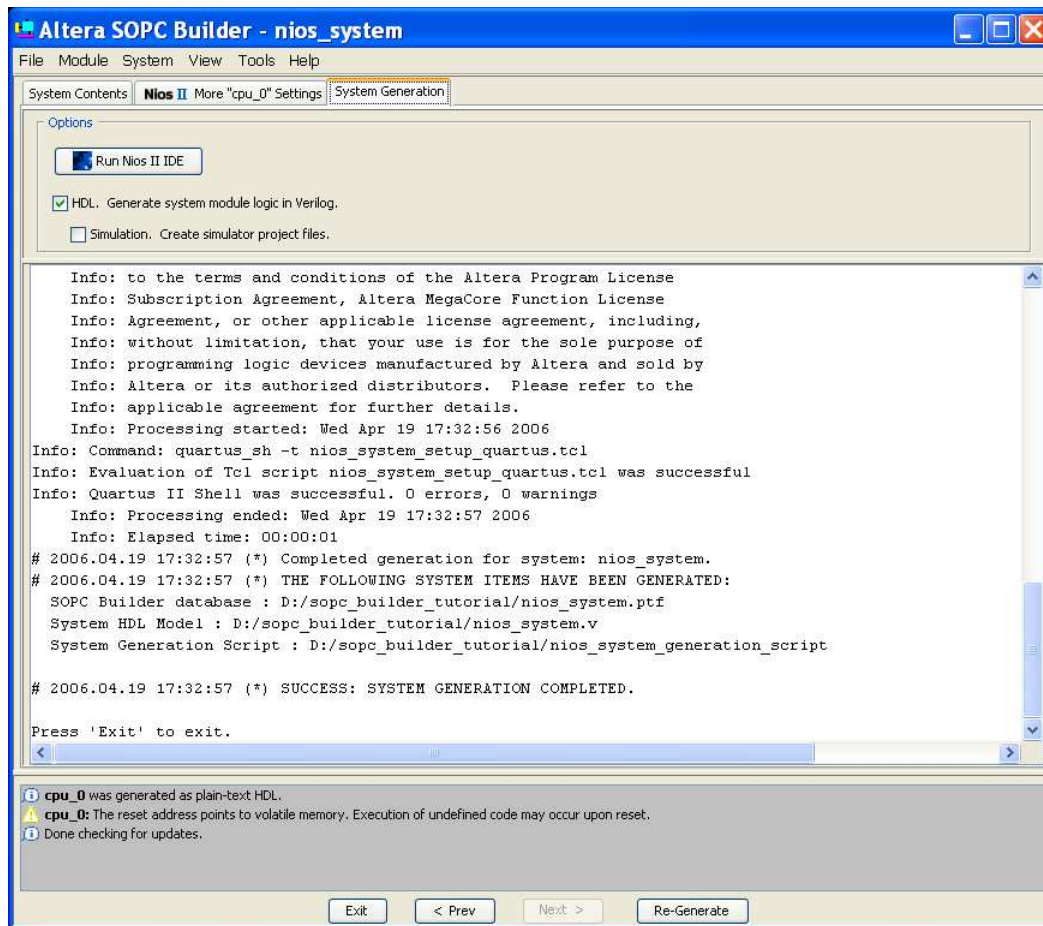


Figure 15. Generation of the system.

Changes to the designed system are easily made at any time by reopening the SOPC Builder tool. Any component in the System Contents tab of the SOPC Builder can be selected and deleted, or a new component can be added and the system regenerated.

3 Integration of the Nios II System into a Quartus II Project

To complete the hardware design, we have to perform the following:

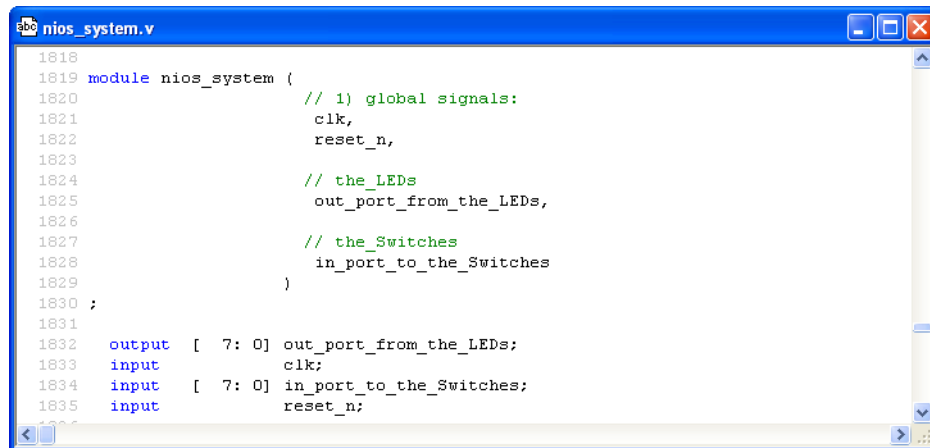
- Instantiate the module generated by the SOPC Builder into the Quartus II project
- Assign the FPGA pins
- Compile the designed circuit
- Program and configure the Cyclone II device on the DE2 board

3.1 Instantiation of the Module Generated by the SOPC Builder

The instantiation of the generated module depends on the design entry method chosen for the overall Quartus II project. We have chosen to use Verilog HDL, but the approach is similar for both VHDL and schematic entry methods.

Normally, the Nios II module is likely to be a part of a larger design. However, in the case of our simple example there is no other circuitry needed. All we need to do is instantiate the Nios II system in our top-level Verilog file, and connect inputs and outputs of the parallel I/O ports, as well as the clock and reset inputs, to the appropriate pins on the Cyclone II device.

The Verilog module generated by the SOPC Builder is in the file *nios_system.v* in the directory of the project. Note that the name of the Verilog module is the same as the system name specified when first using the SOPC Builder. The Verilog code is quite large. Figure 16 depicts the portion of the code that defines the input and output signals for the module *nios_system*. The 8-bit vector that is the input to the parallel port *Switches* is called *in_port_to_the_Switches*. The 8-bit output vector is called *out_port_from_the_LEDs*. The clock and reset signals are called *clk* and *reset_n*, respectively. Note that the reset signal is added automatically by the SOPC Builder; it is called *reset_n* because it is active low.

A screenshot of a text editor window titled 'nios_system.v'. The window displays Verilog code for a module named 'nios_system'. The code includes comments for global signals (clk, reset_n), LEDs (out_port_from_the_LEDs), and switches (in_port_to_the_Switches). It also defines the module's ports: an output for the LEDs, and inputs for the clock, switches, and reset signal.

```
1818
1819 module nios_system (
1820     // 1) global signals:
1821     clk,
1822     reset_n,
1823
1824     // the LEDs
1825     out_port_from_the_LEDs,
1826
1827     // the Switches
1828     in_port_to_the_Switches
1829 )
1830 ;
1831
1832 output [ 7: 0] out_port_from_the_LEDs;
1833 input clk;
1834 input [ 7: 0] in_port_to_the_Switches;
1835 input reset_n;
```

Figure 16. A part of the generated Verilog module.

Figure 17 shows a top-level Verilog module that instantiates the Nios II system. This module is named *lights*, because this is the name we specified in Figure 3 for the top-level design entity in our Quartus II project. Note that the input and output ports of the module use the pin names for the 50-MHz clock, *CLOCK_50*, pushbutton switches, *KEY*, toggle switches, *SW*, and green LEDs, *LEDG*, that are specified in the DE2 User Manual. Type this code into a file called *lights.v*. Add this file and all the *.v files produced by the SOPC Builder to your Quartus II project. Also, add the necessary pin assignments on the DE2 board to your project. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction Using Verilog Design*. Note that an easy way of making the pin assignments when we use the same pin names as in the DE2 User Manual is to import the assignments given in the file called *DE2_pin_assignments.csv* in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages.

Since the system we are designing needs to operate at a 50-MHz clock frequency, add the needed timing assignment in your Quartus II project. The tutorial *Timing Considerations with Verilog-Based Designs* shows how this is done.

```

// Implements a simple Nios II system for the DE2 board.
// Inputs:  SW7–0 are parallel port inputs to the Nios II system
//          CLOCK_50 is the system clock
//          KEY0 is the active-low system reset
// Outputs: LEDG7–0 are parallel port outputs from the Nios II system
module lights (SW, KEY, CLOCK_50, LEDG);
    input  [7:0] SW;
    input  [0:0] KEY;
    input  CLOCK_50;
    output [7:0] LEDG;

// Instantiate the Nios II system module generated by the SOPC Builder:
// nios_system (clk, reset_n, out_port_from_the_LEDs, in_port_to_the_Switches)
    nios_system NiosII (CLOCK_50, KEY[0], LEDG, SW);

endmodule

```

Figure 17. Instantiating the Nios II system.

Having made the necessary settings compile the code. You may see some warning messages associated with the Nios II system, such as some signals being unused or having wrong bit-lengths of vectors; these warnings can be ignored.

3.2 Programming and Configuration

Program and configure the Cyclone II FPGA in the JTAG programming mode as follows:

1. Connect the DE2 board to the host computer by means of a USB cable plugged into the USB-Blaster port. Turn on the power to the DE2 board. Ensure that the RUN/PROG switch is in the RUN position.
2. Select Tools > Programmer to reach the window in Figure 18.
3. If not already chosen by default, select JTAG in the Mode box. Also, if the USB-Blaster is not chosen by default, press the Hardware Setup... button and select the USB-Blaster in the window that pops up.
4. The configuration file *lights.sof* should be listed in the window. If the file is not already listed, then click Add File and select it.
5. Click the box under Program/Configure to select this action.
6. At this point the window settings should appear as indicated in Figure 18. Press Start to configure the FPGA.

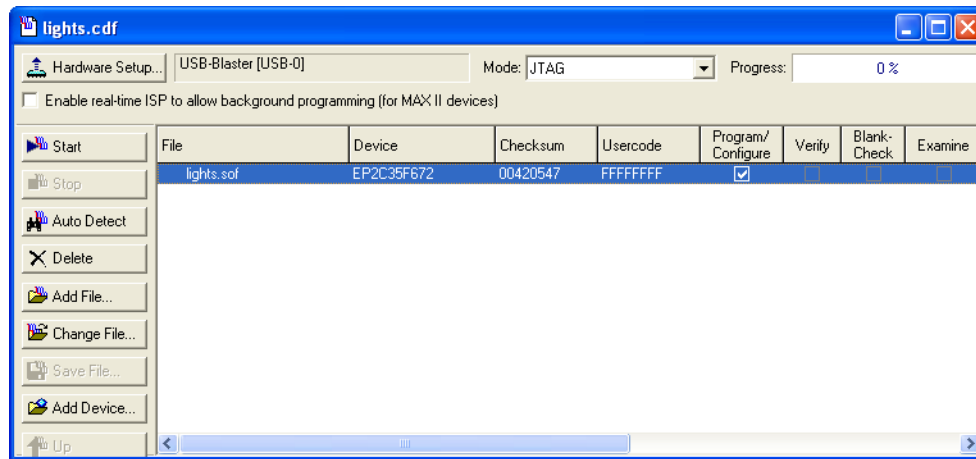


Figure 18. The Programmer window.

4 Running the Application Program

Having configured the required hardware in the FPGA device, it is now necessary to create and execute an application program that performs the desired operation. This can be done by writing the required program either in the Nios II assembly language or in a high-level language such as C. We will illustrate both approaches.

A parallel I/O interface generated by the SOPC Builder is accessible by means of registers in the interface. Depending on how the PIO is configured, there may be as many as four registers. One of these registers is called the Data register. In a PIO configured as an input interface, the data read from the Data register is the data currently present on the PIO input lines. In a PIO configured as an output interface, the data written (by the Nios II processor) into the Data register drives the PIO output lines. If a PIO is configured as a bidirectional interface, then the PIO inputs and outputs use the same physical lines. In this case there is a Data Direction register included, which determines the direction of the input/output transfer. In our unidirectional PIOs, it is only necessary to have the Data register. The addresses assigned by the SOPC Builder are 0x00001800 for the Data register in the PIO called Switches and 0x00001810 for the Data register in the PIO called LEDs, as indicated in Figure 14.

You can find a full description of the PIO interface by opening the SOPC Builder window in Figure 14 and right-clicking on the module name of a PIO (either Switches or LEDs). Then, in the pop-up box select *Data Sheet* to open the document *PIO Core with Avalon Interface* which gives a full description of the interface. To use this facility you need to be connected to the Internet.

4.1 Using a Nios II Assembly Language Program

Figure 19 gives a Nios II assembly-language program that implements our trivial task. The program loads the addresses of the Data registers in the two PIOs into processor registers *r2* and *r3*. It then has an infinite loop that merely transfers the data from the input PIO, *Switches*, to the output PIO, *LEDs*. Note that the program contains a statement that includes the *nios_macros*, and two statements, *GFUNC* and *BREAK*, needed to assemble the program properly.

```

.include "nios_macros.s"

.equ Switches, 0x00001800
.equ LEDs, 0x00001810

GFUNC _start

        movia    r2, Switches
        movia    r3, LEDs
loop:   ldbio     r4, 0(r2)
        stbio     r4, 0(r3)
        br        loop

        BREAK

```

Figure 19. Assembly language code to control the lights.

Enter this code into a file *lights.s* and place the file into a working directory. We placed the file into directory *sopc_builder_tutorial\app_software*. The program has to be assembled and converted into an S-Record file, *lights.srec*, suitable for downloading into the implemented Nios II system.

Altera provides the *monitor* software, called *Nios II Debug Client*, for use with the DE2 board. This software provides a simple means for compiling, assembling and downloading of programs into a Nios II system implemented on a DE2 board. It also makes it possible for the user to perform debugging tasks. A description of this software is available in the *Nios II Debug Client* tutorial.

Open the Nios II Debug Client, which leads to the window in Figure 20. This software needs to know the characteristics of the designed Nios II system, which are given in the ptf file *lights.ptf*. Select the file *lights.ptf* as indicated in the figure. Note that this file is in the design directory *sopc_builder_tutorial*. The Nios II Debug Client also needs to know where to load the application program. In our case, this is the memory block in the FPGA device. The SOPC builder assigned the name *onchip_memory_0* to this block. If not already done, select this name in the window in Figure 20. Having provided the necessary information, click **Confirm**.

Next, the main Nios II Debug Client window appears, as shown in Figure 21. To assemble and download the *light.s* program click **Compile & Load**. A dialog box in Figure 22 appears. Select the file *lights.s* as indicated in the figure and click **Open**.

As a result of opening the file, the Nios II Debug Client invokes an assembler program, followed by a linker program. The commands used to invoke these programs, and the output they produce, can be viewed in the **Debug** tab of the Client window. The downloaded program is displayed in the **Disassemble** tab of the Client window as illustrated in Figure 23. Observe that **movia** is a *pseudoinstruction* which is implemented as two separate instructions. See the *Nios II Processor Reference Handbook* for a description of the Nios II instruction set.

Click **Run** to execute the program. With the program running, you can now test the design by turning the switches, *SW7* to *SW0* on and off; the LEDs should respond accordingly.

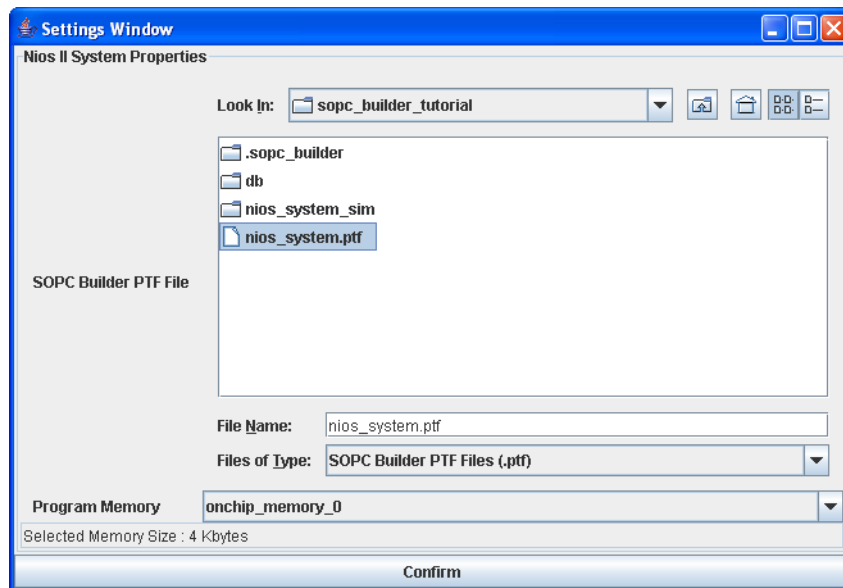


Figure 20. The Nios II Debug Client Settings window.

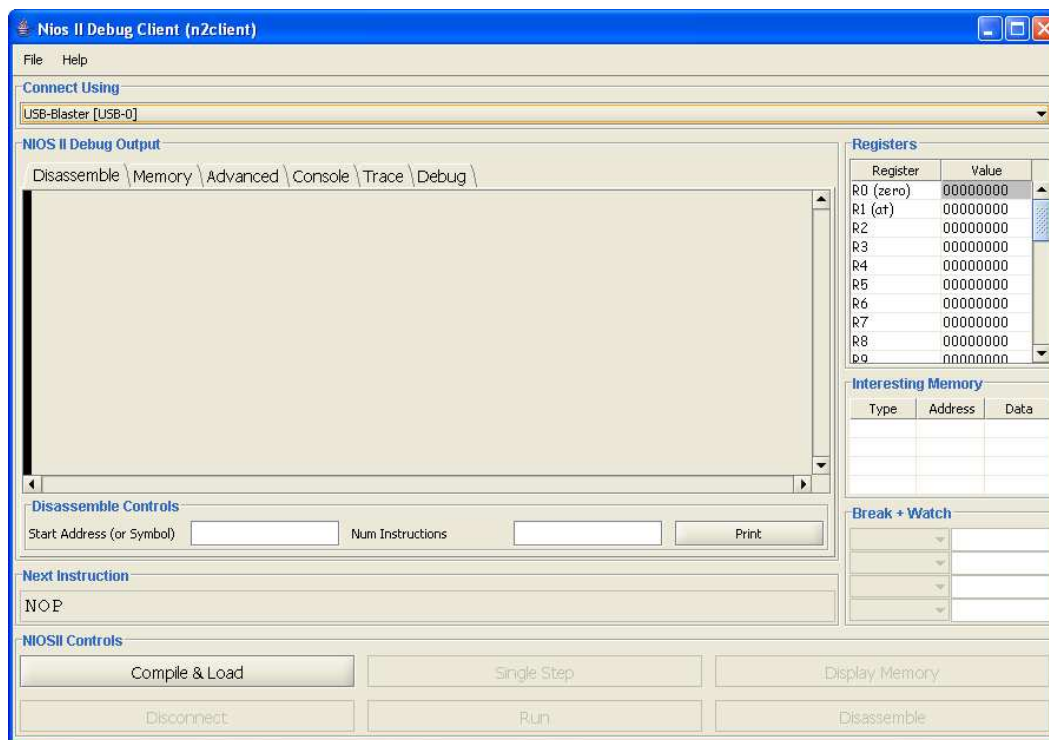


Figure 21. The Nios II Debug Client window.

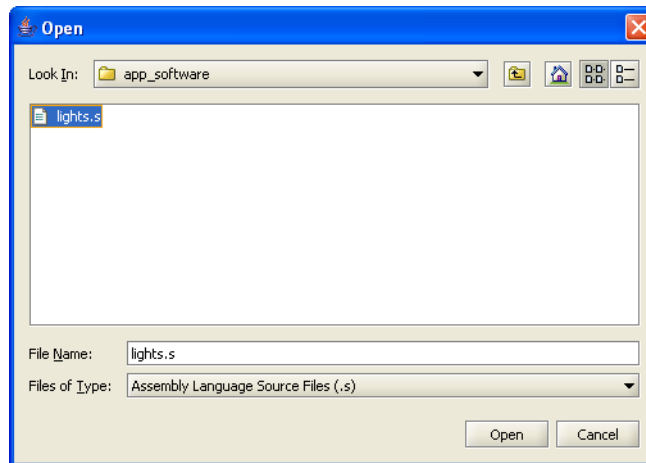


Figure 22. Open File dialog box.

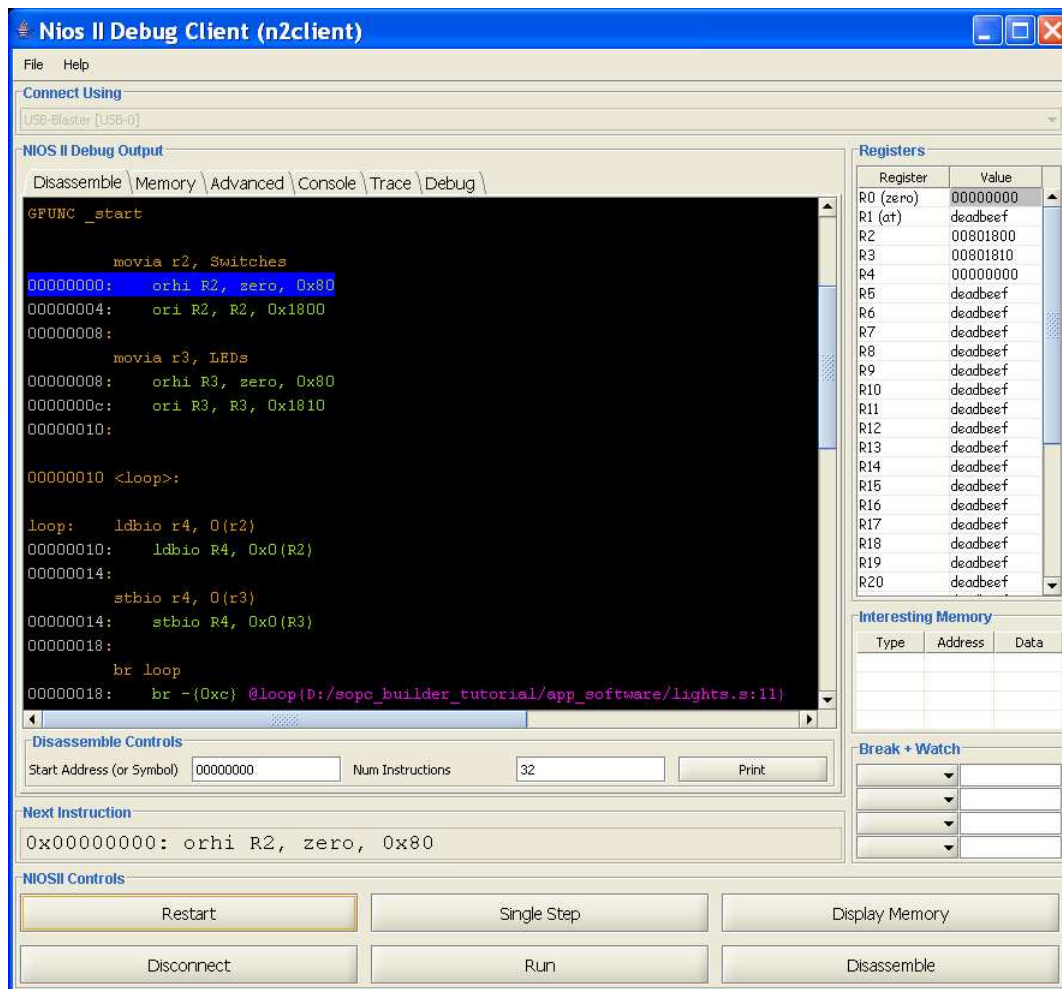


Figure 23. Display of the downloaded program.

The Nios II Debug Client allows a number of useful functions to be performed in a simple manner. They include:

- single stepping through the program
- examining the contents of processor registers
- examining the contents of the memory
- setting breakpoints for debugging purposes
- disassembling the downloaded program

A description of this software and all of its features is available in the *Nios II Debug Client* tutorial.

4.2 Using a C-Language Program

An application program written in the C language can be handled in the same way as the assembly-language program. A C program that implements our simple task is given in Figure 24. Enter this code into a file called *lights.c*.

```
#define Switches (volatile char *) 0x0001800
#define LEDs (char *) 0x0001810

void main()
{   while (1)
    *LEDs = *Switches;
}
```

Figure 24. C language code to control the lights.

To use this program, get to the window in Figure 21 and press **Compile & Load**. In the dialog box in Figure 22 select the file *lights.c*. The rest of the operation is the same as described above.

Copyright ©2006 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.

Using the SDRAM Memory on Altera's DE2 Board with Verilog Design

This tutorial explains how the SDRAM chip on Altera's DE2 Development and Education board can be used with a Nios II system implemented by using the Altera SOPC Builder. The discussion is based on the assumption that the reader has access to a DE2 board and is familiar with the material in the tutorial *Introduction to the Altera SOPC Builder Using Verilog Design*.

The screen captures in the tutorial were obtained using the Quartus® II version 5.1; if other versions of the software are used, some of the images may be slightly different.

Contents:

Example Nios II System

The SDRAM Interface

Using the SOPC Builder to Generate the Nios II System

Integration of the Nios II System into the Quartus II Project

Using a Phase-Locked Loop

The introductory tutorial *Introduction to the Altera SOPC Builder Using Verilog Design* explains how the memory in the Cyclone II FPGA chip can be used in the context of a simple Nios II system. For practical applications it is necessary to have a much larger memory. The Altera DE2 board contains an SDRAM chip that can store 8 Mbytes of data. This memory is organized as 1M x 16 bits x 4 banks. The SDRAM chip requires careful timing control. To provide access to the SDRAM chip, the SOPC Builder implements an *SDRAM Controller* circuit. This circuit generates the signals needed to deal with the SDRAM chip.

1 Example Nios II System

As an illustrative example, we will add the SDRAM to the Nios II system described in the *Introduction to the Altera SOPC Builder Using Verilog Design* tutorial. Figure 1 gives the block diagram of our example system.

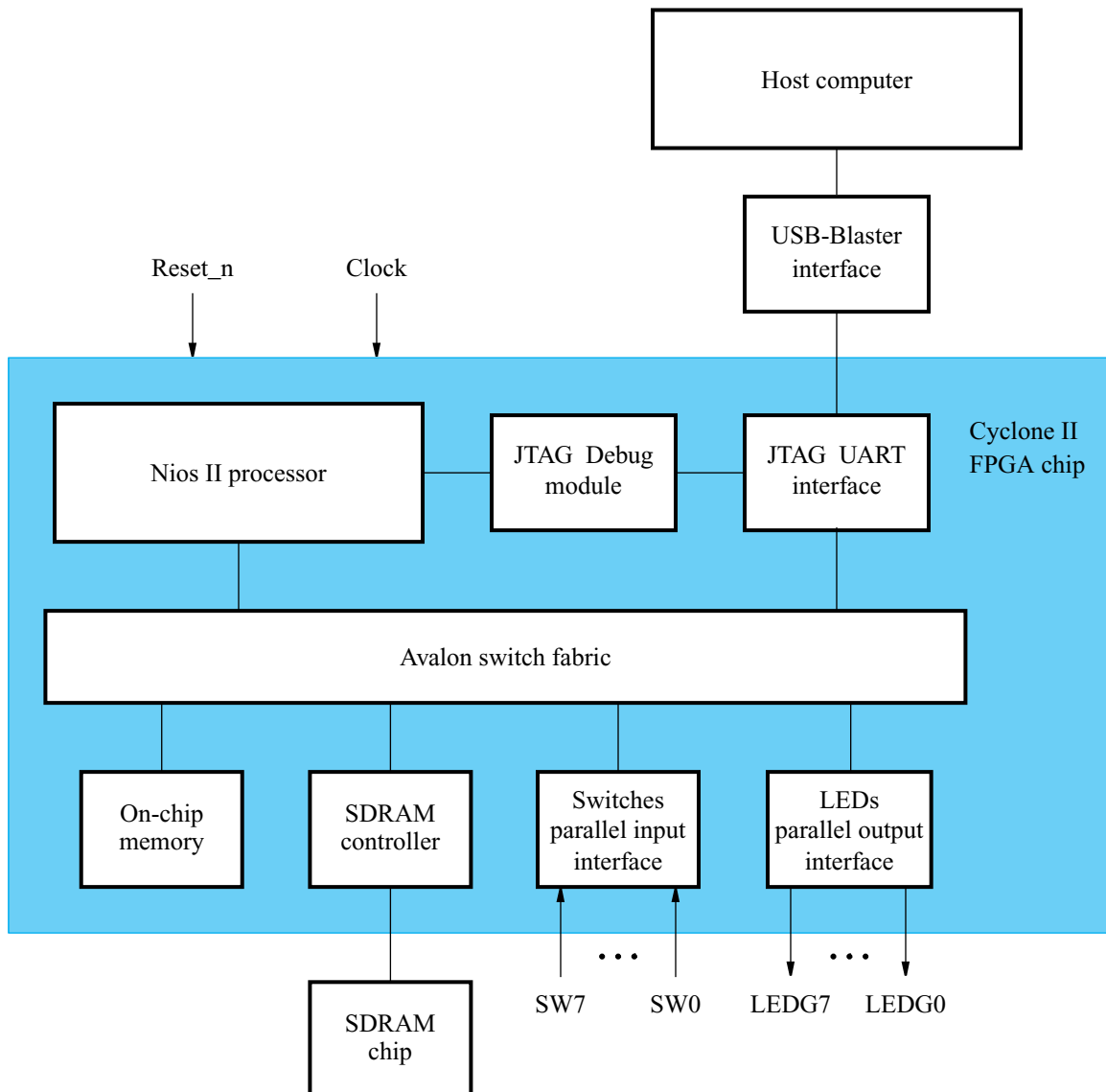


Figure 1. Example Nios II system implemented on the DE2 board.

The system realizes a trivial task. Eight toggle switches on the DE2 board, $SW7 - 0$, are used to turn on or off the eight green LEDs, $LEDG7 - 0$. The switches are connected to the Nios II system by means of a parallel I/O

interface configured to act as an input port. The LEDs are driven by the signals from another parallel I/O interface configured to act as an output port. To achieve the desired operation, the eight-bit pattern corresponding to the state of the switches has to be sent to the output port to activate the LEDs. This will be done by having the Nios II processor execute an application program. Continuous operation is required, such that as the switches are toggled the lights change accordingly.

The introductory tutorial showed how we can use the SOPC Builder to design the hardware needed to implement this task, assuming that the application program which reads the state of the toggle switches and sets the green LEDs accordingly is loaded into a memory block in the FPGA chip. In this tutorial, we will explain how the SDRAM chip on the DE2 board can be included in the system in Figure 1, so that our application program can be run from the SDRAM rather than from the on-chip memory.

Doing this tutorial, the reader will learn about:

- Using the SOPC Builder to include an SDRAM interface for a Nios II-based system
- Timing issues with respect to the SDRAM on the DE2 board
- Using a phase-locked loop (PLL) to control the clock timing

2 The SDRAM Interface

The SDRAM chip on the DE2 board has the capacity of 64 Mbits (8 Mbytes). It is organized as 1M x 16 bits x 4 banks. The signals needed to communicate with this chip are shown in Figure 2. All of the signals, except the clock, can be provided by the SDRAM Controller that can be generated by using the SOPC Builder. The clock signal is provided separately. It has to meet the clock-skew requirements as explained in section 5. Note that some signals are active low, which is denoted by the suffix N.

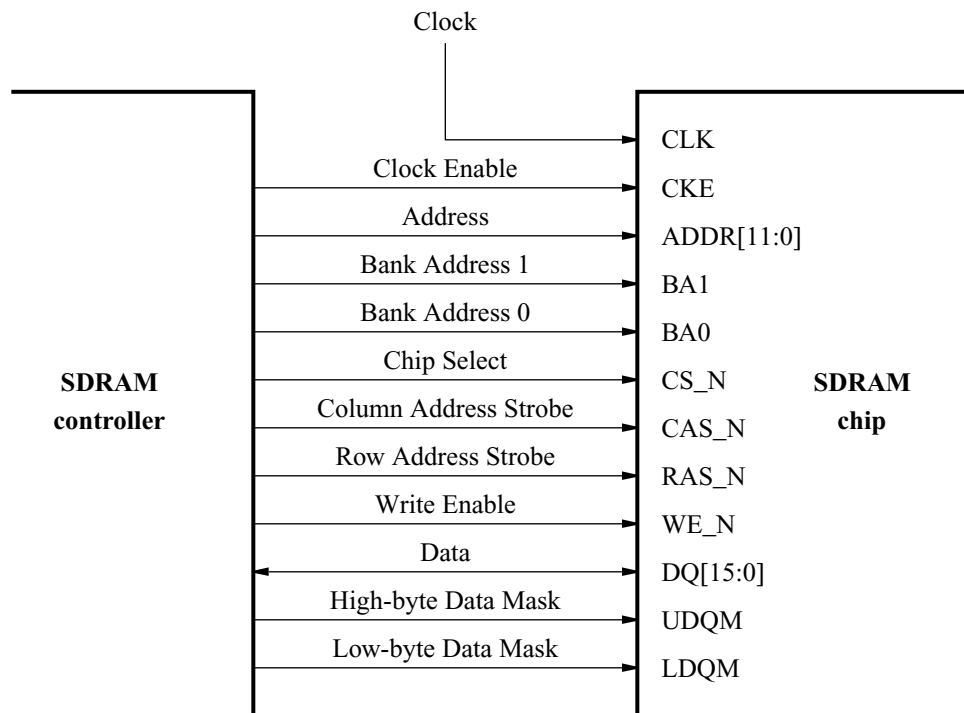


Figure 2. The SDRAM signals.

3 Using the SOPC Builder to Generate the Nios II System

Our starting point will be the Nios II system discussed in the *Introduction to the Altera SOPC Builder Using Verilog Design* tutorial, which we implemented in a project called *lights*. We specified the system shown in Figure 3.

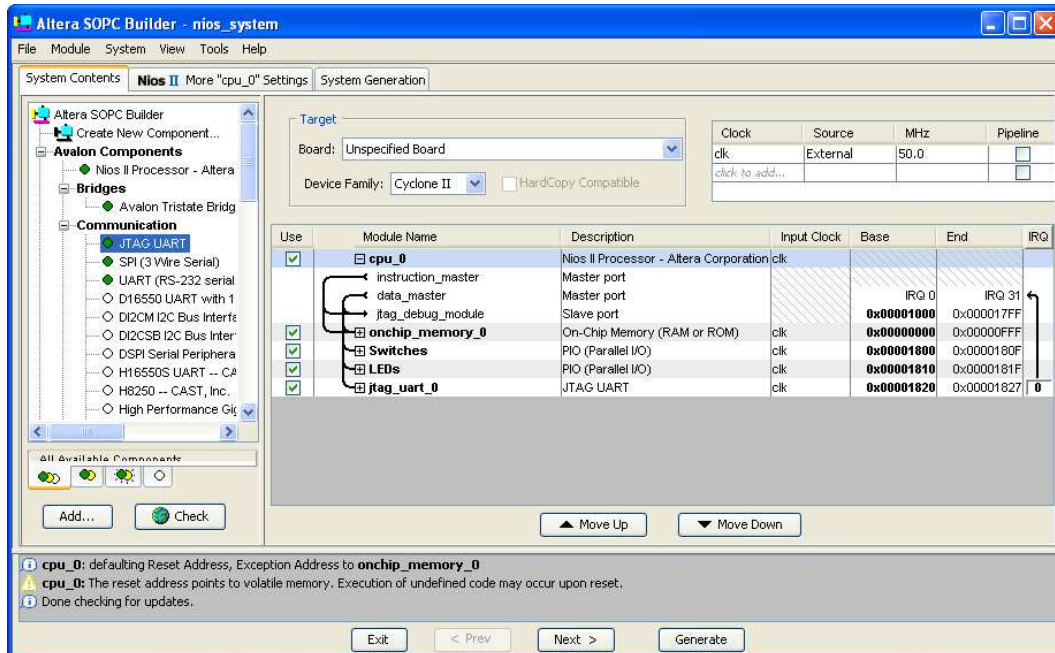


Figure 3. The Nios II system defined in the introductory tutorial.

If you saved the *lights* project, then open this project in the Quartus II software and then open the SOPC Builder. Otherwise, you need to create and implement the project, as explained in the introductory tutorial, to obtain the system shown in the figure.

To add the SDRAM, in the window of Figure 3 select **Avalon Components > Memory > SDRAM Controller** and click **Add**. A window depicted in Figure 4 appears. Set the Data Width parameter to 16 bits and leave the default values for the rest. Since we will not simulate the system in this tutorial, do not select the option **Include a functional memory model in the system testbench**. Click **Finish**. Now, in the window of Figure 3, there will be an **sdram_0** module added to the design. Since there is only one SDRAM on the DE2 board, change the name of this module to simply **sdram**. Then, the expanded system is defined as indicated in Figure 5. Observe that the SOPC Builder assigned the base address 0x00800000 to the SDRAM. Leave the addresses of all modules as assigned in the figure and regenerate the system.

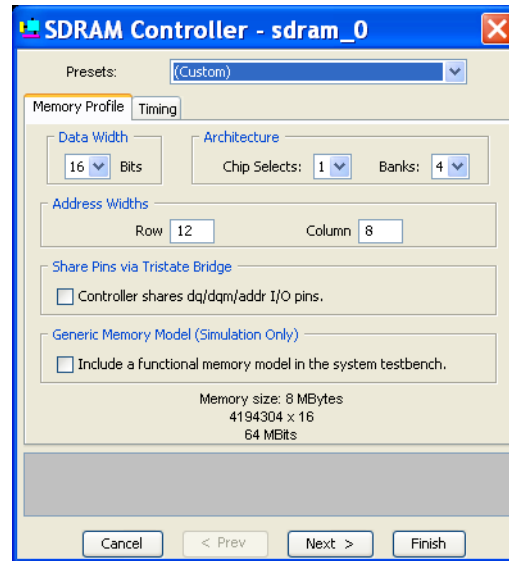


Figure 4. Add the SDRAM Controller.

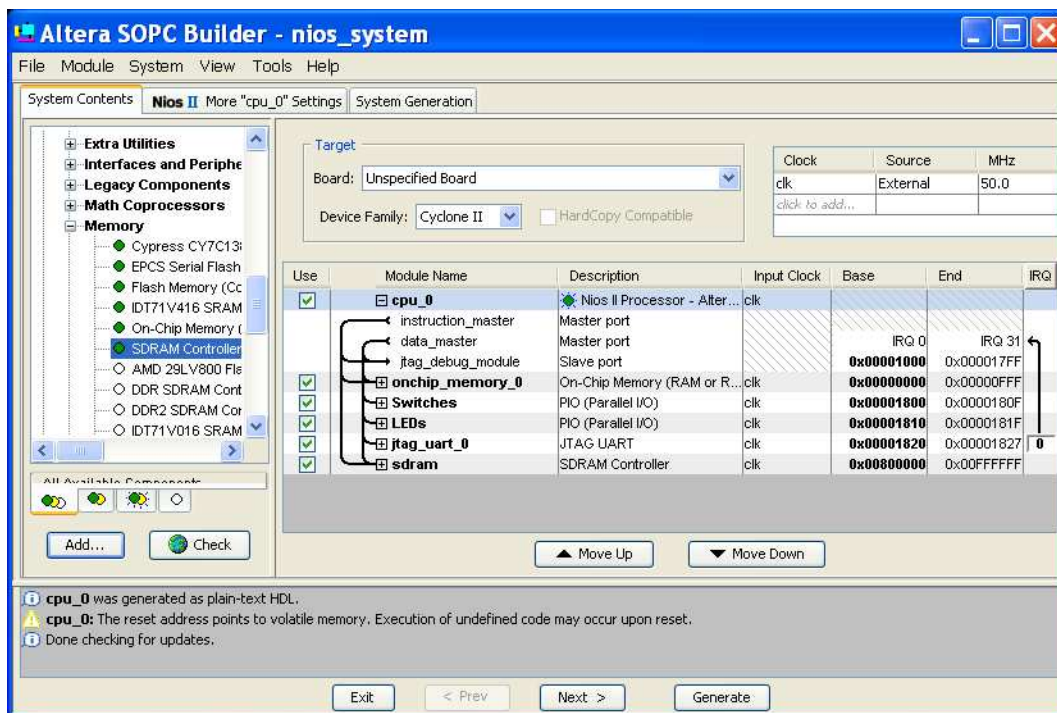


Figure 5. The expanded Nios II system.

The augmented Verilog module generated by the SOPC Builder is in the file *nios_system.v* in the directory of the project. Figure 6 depicts the portion of the code that defines the input and output signals for the module *nios_system*. As in our initial system that we developed in the introductory tutorial, the 8-bit vector that is the input to the parallel port *Switches* is called *in_port_to_the_Switches*. The 8-bit output vector is called *out_port_from_the_LEDs*. The clock and reset signals are called *clk* and *reset_n*, respectively. A new module, called *sdr0*, is included. It involves the signals indicated in Figure 2. For example, the address lines are referred to as the **output** vector *zs_addr_from_the_sdr0[11:0]*. The data lines are referred to as the **inout** vector *zs_dq_to_and_from_the_sdr0[15:0]*. This is a vector of the **inout** type because the data lines are bidirectional.

```

3056
3057 module nios_system (
3058     // 1) global signals:
3059     clk,
3060     reset_n,
3061
3062     // the LEDs
3063     out_port_from_the_LEDs,
3064
3065     // the Switches
3066     in_port_to_the_Switches,
3067
3068     // the sdram
3069     zs_addr_from_the_sdram,
3070     zs_ba_from_the_sdram,
3071     zs_cas_n_from_the_sdram,
3072     zs_cke_from_the_sdram,
3073     zs_cs_n_from_the_sdram,
3074     zs_dq_to_and_from_the_sdram,
3075     zs_dqm_from_the_sdram,
3076     zs_ras_n_from_the_sdram,
3077     zs_we_n_from_the_sdram
3078 )
3079 ;
3080
3081 output [ 7: 0] out_port_from_the_LEDs;
3082 output [ 11: 0] zs_addr_from_the_sdram;
3083 output [ 1: 0] zs_ba_from_the_sdram;
3084 output zs_cas_n_from_the_sdram;
3085 output zs_cke_from_the_sdram;
3086 output zs_cs_n_from_the_sdram;
3087 inout [ 15: 0] zs_dq_to_and_from_the_sdram;
3088 output [ 1: 0] zs_dqm_from_the_sdram;
3089 output zs_ras_n_from_the_sdram;
3090 output zs_we_n_from_the_sdram;
3091 input clk;

```

Figure 6. A part of the generated Verilog module.

4 Integration of the Nios II System into the Quartus II Project

Now, we have to instantiate the expanded Nios II system in the top-level Verilog module, as we have done in the tutorial *Introduction to the Altera SOPC Builder Using Verilog Design*. The module is named *lights*, because this is the name of the top-level design entity in our Quartus II project.

A first attempt at creating the new module is presented in Figure 7. The input and output ports of the module use the pin names for the 50-MHz clock, *CLOCK_50*, pushbutton switches, *KEY*, toggle switches, *SW*, and green LEDs, *LEDG*, as used in our original design. They also use the pin names *DRAM_CLK*, *DRAM_CKE*, *DRAM_ADDR*, *DRAM_BA_1*, *DRAM_BA_0*, *DRAM_CS_N*, *DRAM_CAS_N*, *DRAM_RAS_N*, *DRAM_WE_N*, *DRAM_DQ*, *DRAM_UDQM*, and *DRAM_LDQM*, which correspond to the SDRAM signals indicated in Figure 2. All of these names are those specified in the DE2 User Manual, which allows us to make the pin assignments by importing them from the file called *DE2_pin_assignments.csv* in the directory *DE2_tutorials\design_files*, which is included on the CD-ROM that accompanies the DE2 board and can also be found on Altera's DE2 web pages.

Observe that the two *Bank Address* signals are treated by the SOPC Builder as a two-bit vector called *zs_ba_from_the_sdram[1:0]*, as seen in Figure 6. However, in the *DE2_pin_assignments.csv* file these signals are given as scalars *DRAM_BA_1* and *DRAM_BA_0*. Therefore, in our Verilog module, we concatenated these signals as *{DRAM_BA_1, DRAM_BA_0}*. Similarly, the vector *zs_dqm_from_the_sdram[1:0]* corresponds to *{DRAM_UDQM, DRAM_LDQM}*.

Finally, note that we tried an obvious approach of using the 50-MHz system clock, *CLOCK_50*, as the clock signal, *DRAM_CLK*, for the SDRAM chip. This is specified by the **assign** statement in the code. This approach leads to a potential timing problem caused by the clock skew on the DE2 board, which can be fixed as explained in section 5.

```

// Implements the augmented Nios II system for the DE2 board.
// Inputs:  SW7–0 are parallel port inputs to the Nios II system.
//          CLOCK_50 is the system clock.
//          KEY0 is the active-low system reset.
// Outputs: LEDG7–0 are parallel port outputs from the Nios II system.
//          SDRAM ports correspond to the signals in Figure 2; their names are those
//          used in the DE2 User Manual.
module lights (SW, KEY, CLOCK_50, LEDG, DRAM_CLK, DRAM_CKE,
    DRAM_ADDR, DRAM_BA_1, DRAM_BA_0, DRAM_CS_N, DRAM_CAS_N, DRAM_RAS_N,
    DRAM_WE_N, DRAM_DQ, DRAM_UDQM, DRAM_LDQM);
    input [7:0] SW;
    input [0:0] KEY;
    input CLOCK_50;
    output [7:0] LEDG;
    output [11:0] DRAM_ADDR;
    output DRAM_BA_1, DRAM_BA_0, DRAM_CAS_N, DRAM_RAS_N, DRAM_CLK;
    output DRAM_CKE, DRAM_CS_N, DRAM_WE_N, DRAM_UDQM, DRAM_LDQM;
    inout [15:0] DRAM_DQ;

// Instantiate the Nios II system module generated by the SOPC Builder
    nios_system NiosII (
        CLOCK_50,
        KEY[0],
        LEDG,
        SW,
        DRAM_ADDR,
        {DRAM_BA_1, DRAM_BA_0},
        DRAM_CAS_N,
        DRAM_CKE,
        DRAM_CS_N,
        DRAM_DQ,
        {DRAM_UDQM, DRAM_LDQM},
        DRAM_RAS_N,
        DRAM_WE_N);
    assign DRAM_CLK = CLOCK_50;

endmodule

```

Figure 7. A first attempt at instantiating the expanded Nios II system.

As an experiment, you can enter the code in Figure 7 into a file called *lights.v*. Add this file and all the *.v files produced by the SOPC Builder to your Quartus II project. Compile the code and download the design into the Cyclone II FPGA on the DE2 board. Use the application program from the tutorial *Introduction to the Altera SOPC Builder*, which is shown in Figure 8.

```

.include "nios_macros.s"

.equ Switches, 0x00001800
.equ LEDs, 0x00001810

GFUNC _start

        movia    r2, Switches
        movia    r3, LEDs
loop:   ldbio     r4, 0(r2)
        stbio     r4, 0(r3)
        br       loop

        BREAK

```

Figure 8. Assembly language code to control the lights.

Use the Nios II Debug Client, which is described in the tutorial *Nios II Debug Client*, to assemble, download, and run this application program. If successful, the lights on the DE2 board will respond to the operation of the toggle switches.

Due to the clock skew problem mentioned above, the Nios II processor may be unable to properly access the SDRAM chip. A possible indication of this may be given by the Nios II Debug Client, which may display the message depicted in Figure 9. To solve the problem, it is necessary to modify the design as indicated in the next section.

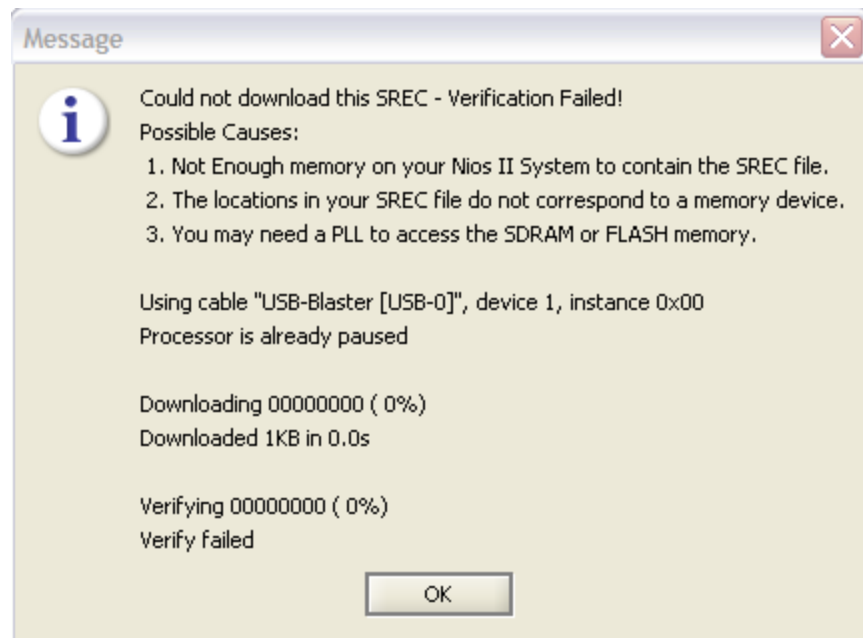


Figure 9. An error message.

5 Using a Phase-Locked Loop

The clock skew depends on physical characteristics of the DE2 board. For proper operation of the SDRAM chip, it is necessary that its clock signal, *DRAM_CLK*, leads the Nios II system clock, *CLOCK_50*, by 3 nanoseconds. This can be accomplished by using a *phase-locked loop (PLL)* circuit. There exists a Quartus II Megafunction, called *ALTPLL*, which can be used to generate the desired circuit. The circuit can be created, by using the Quartus II MegaWizard Plug-In Manager, as follows:

1. Select Tools > MegaWizard Plug-In Manager. This leads to the window in Figure 10. Choose the action Create a new custom megafunction variation and click Next.

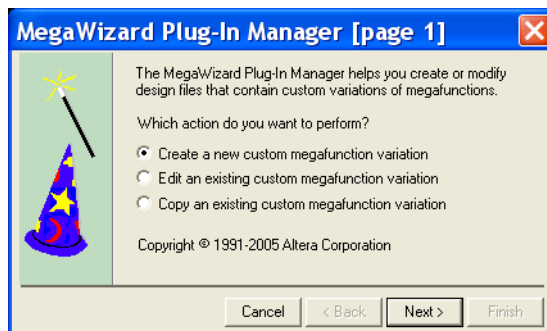


Figure 10. The MegaWizard.

2. In the window in Figure 11, specify that Cyclone II is the device family used and that the circuit should be defined in Verilog HDL. Also, specify that the generated output (Verilog) file should be called *sdram_pll.v*. From the list of megafunctions in the left box select I/O > ALTPLL. Click Next.

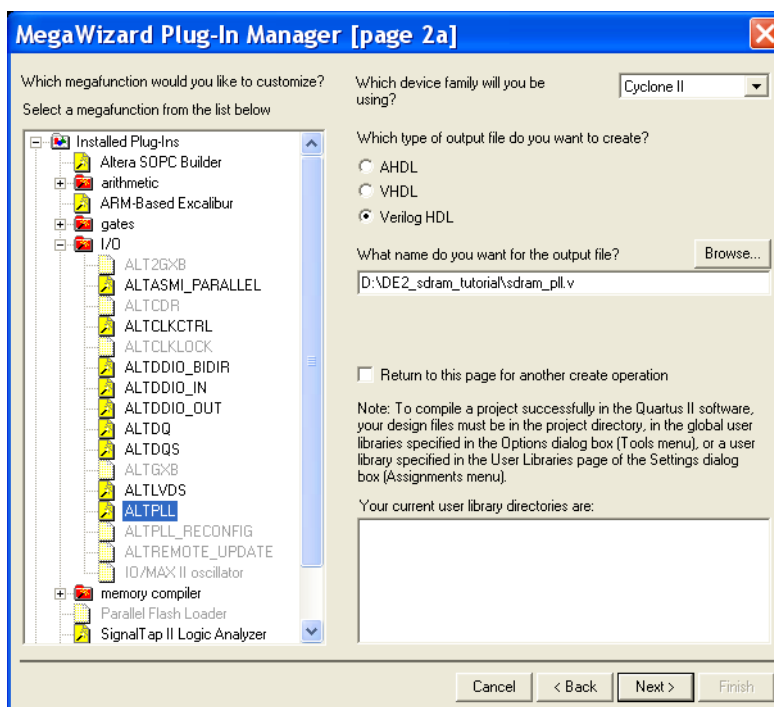


Figure 11. Select the megafunction and name the output file.

3. In Figure 12, specify that the frequency of the *inclk0* input is 50 MHz. Leave the other parameters as given by default. Click Next to reach the window in Figure 13.

The screenshot shows the 'MegaWizard Plug-In Manager - ALTPLL [page 3 of 9]' window. On the left, a block diagram of the 'sdram_pll' component is shown with inputs 'inclk0' and 'areset', and outputs 'c0' and 'locked'. The component is labeled 'Cyclone II'. The right pane shows the 'General' tab configuration:

- General**
 - Which device family will you be using? Cyclone II
 - Which device speed grade will you be using? Any
 - What is the frequency of the inclk0 input? 50 MHz
 - ☐ Set up PLL in LVDS mode
 - Data rate: Not Available Mbps
- PLL type**
 - Which PLL type will you be using?
 - ☐ Fast PLL
 - ☐ Enhanced PLL
 - ☒ Select the PLL type automatically
- Operation mode**
 - How will the PLL outputs be generated?
 - ☒ Use the feedback path inside the PLL
 - ☒ In Normal Mode
 - ☐ In Source-Synchronous Compensation Mode
 - ☐ In Zero Delay Buffer Mode
 - ☐ With no compensation
 - ☐ Create an 'fbn' input for an external feedback (External Feedback Mode)
 - Which output clock will be compensated for? c0

Buttons at the bottom: Documentation, Cancel, < Back, Next >, Finish.

Figure 12. Define the clock frequency.

The screenshot shows the 'MegaWizard Plug-In Manager - ALTPLL [page 4 of 9]' window. The left pane is identical to Figure 12. The right pane shows the 'Scan/Lock' tab configuration:

- Dynamic configuration**
 - ☐ Create optional inputs for dynamic reconfiguration
- Optional inputs**
 - ☐ Create an 'pllena' input to selectively enable the PLL
 - ☐ Create an 'areset' input to asynchronously reset the PLL
 - ☐ Create an 'prdena' input to selectively enable the phase/freq. detector
- Lock output**
 - ☐ Create 'locked' output
 - ☐ Hold 'locked' output low 1048575 cycles after the PLL initializes
- Advanced PLL Parameters**
 - Using these parameters is recommended for advanced users only
 - ☐ Create output file(s) using the 'Advanced' PLL parameters
 - Configurations with output clock(s) that use cascade counters are not supported

Buttons at the bottom: Documentation, Cancel, < Back, Next >, Finish.

Figure 13. Remove unnecessary signals.

4. We are interested only in the input signal *inclk0* and the output signal *c0*. Remove the other two signals shown in the block diagram in the figure by de-selecting the optional input *areset* as well as the locked output, as indicated in the figure. Click Next on this page as well as on page 5, until you reach page 6 which is shown in Figure 14.

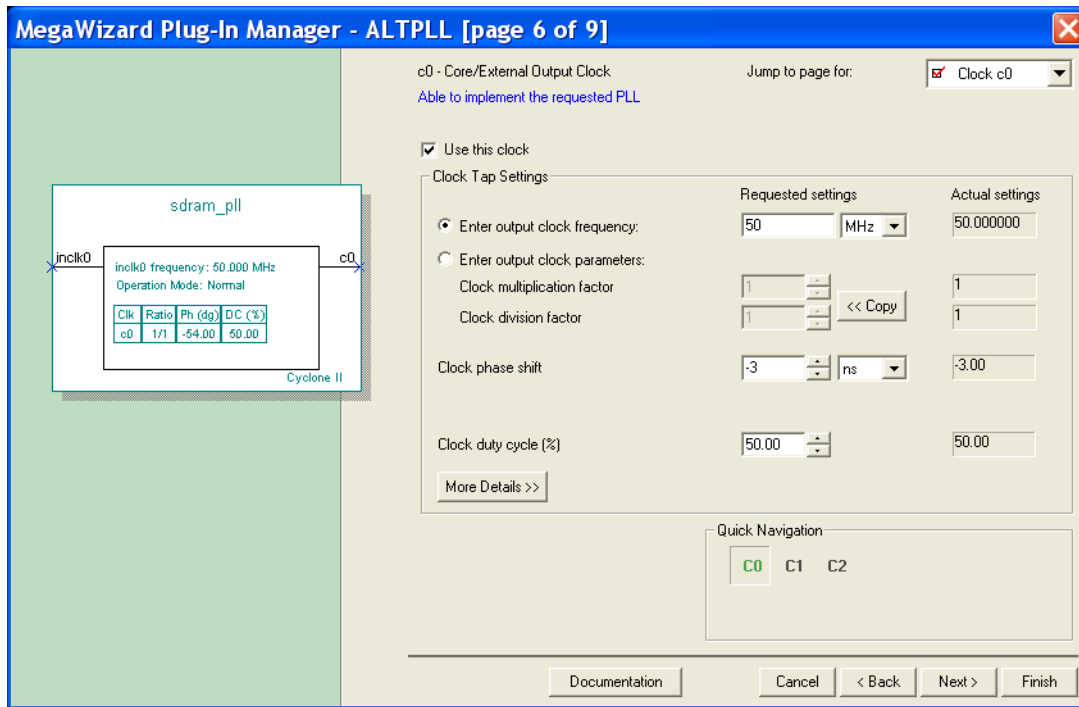


Figure 14. Specify the phase shift.

5. The shifted clock signal is called *c0*. Specify that the output clock frequency is 50 MHz. Also, specify that a phase shift of -3 ns is required, as indicated in the figure. Click Finish, which advances to page 9.
6. In the summary window in Figure 15 click Finish to complete the process.

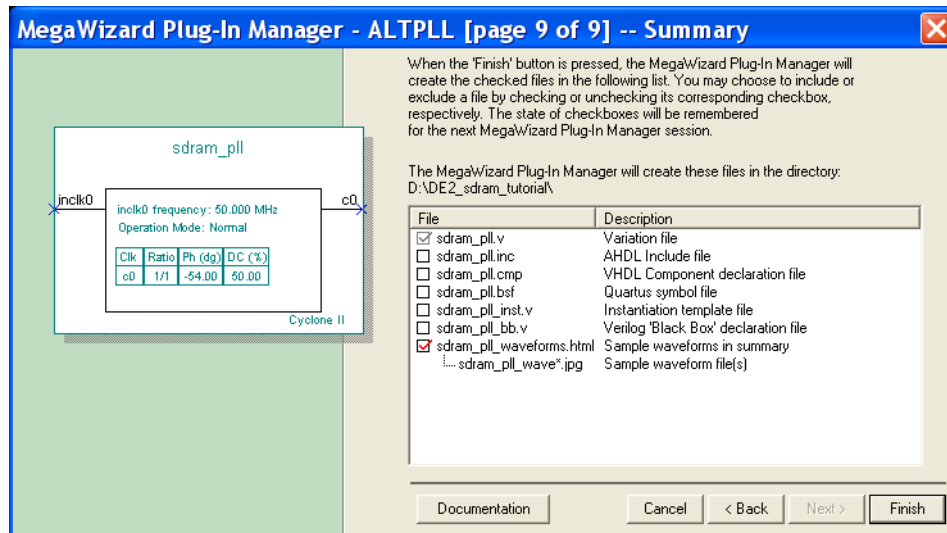


Figure 15. The summary page.

The desired PLL circuit is now defined as a Verilog module in the file *sdr_pll.v*, which is placed in the project directory. Add this file to the *lights* project. Figure 16 shows the module ports, consisting of signals *inclk0* and *c0*.

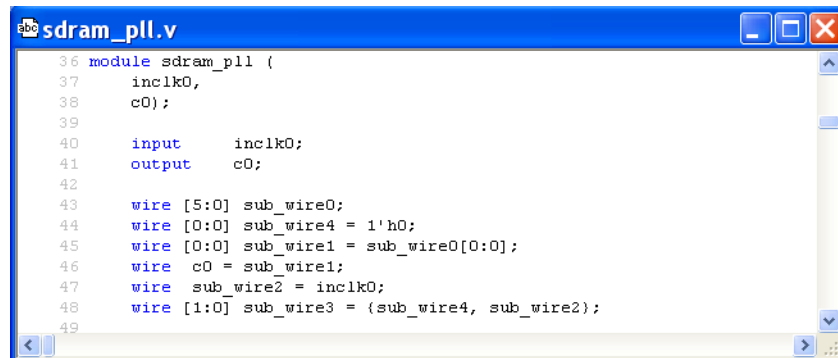


Figure 16. The generated PLL module.

Next, we have to fix the top-level Verilog module, given in Figure 7, to include the PLL circuit. The desired code is shown in Figure 17. The PLL circuit connects the shifted clock output *c0* to the pin *DRAM_CLK*.

```

// Implements the augmented Nios II system for the DE2 board.
// Inputs:  SW7–0 are parallel port inputs to the Nios II system.
//         CLOCK_50 is the system clock.
//         KEY0 is the active-low system reset.
// Outputs: LEDG7–0 are parallel port outputs from the Nios II system.
//         SDRAM ports correspond to the signals in Figure 2; their names are those
//         used in the DE2 User Manual.
module lights (SW, KEY, CLOCK_50, LEDG, DRAM_CLK, DRAM_CKE,
    DRAM_ADDR, DRAM_BA_1, DRAM_BA_0, DRAM_CS_N, DRAM_CAS_N, DRAM_RAS_N,
    DRAM_WE_N, DRAM_DQ, DRAM_UDQM, DRAM_LDQM);
    input [7:0] SW;
    input [0:0] KEY;
    input CLOCK_50;
    output [7:0] LEDG;
    output [11:0] DRAM_ADDR;
    output DRAM_BA_1, DRAM_BA_0, DRAM_CAS_N, DRAM_RAS_N, DRAM_CLK;
    output DRAM_CKE, DRAM_CS_N, DRAM_WE_N, DRAM_UDQM, DRAM_LDQM;
    inout [15:0] DRAM_DQ;

// Instantiate the Nios II system module generated by the SOPC Builder
    nios_system NiosII (
        CLOCK_50,
        KEY[0],
        LEDG,
        SW,
        DRAM_ADDR,
        {DRAM_BA_1, DRAM_BA_0},
        DRAM_CAS_N,
        DRAM_CKE,
        DRAM_CS_N,
        DRAM_DQ,
        {DRAM_UDQM, DRAM_LDQM},
        DRAM_RAS_N,
        DRAM_WE_N);

// Instantiate the module sdram_pll (inclk0, c0)
    sdram_pll neg_3ns (CLOCK_50, DRAM_CLK);

endmodule

```

Figure 17. Proper instantiation of the expanded Nios II system.

Compile the code and download the design into the Cyclone II FPGA on the DE2 board. Use the application program in Figure 8 to test the circuit.

Copyright ©2006 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.